FiPy
A Finite Volume PDE Solver Using Python

Daniel Wheeler
Jonathan E. Guyer
James A. Warren

Metallurgy Division
and the Center for Theoretical and Computational Materials Science
Materials Science and Engineering Laboratory

November 5, 2004

Version 0.1

National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

Contents

Part I

Introduction

Introduction Contents

Chapter 1

Overview

FiPy is an object oriented, partial differential equation (PDE) solver, written in Python [1], based on a standard finite volume (FV) approach. The framework has been developed in the Metallurgy Division and Center for Theoretical and Computational Materials Science (CTCMS), in the Materials Science and Engineering Laboratory (MSEL) at the National Institute of Standards and Technology (NIST).

The solution of coupled sets of PDEs is ubiquitous to the numerical simulation of science problems. Numerous PDE solvers exist, using a variety of languages and numerical approaches. Many are proprietary, expensive and difficult to customize. As a result, scientists spend considerable resources repeatedly developing limited tools for specific problems. Our approach, combining the FV method and Python, provides a tool that is extensible, powerful and freely available. A significant advantage to Python is the existing suite of tools for array calculations, sparse matrices and data rendering.

The FiPy framework includes terms for transient diffusion, convection and standard sources, enabling the solution of arbitrary combinations of coupled elliptic, hyperbolic and parabolic PDEs. Currently implemented models include phase field [2, 3] treatments of polycrystalline, dendritic, and electrochemical phase transformations as well as a level set treatment of the electrodeposition process [4].

The latest information about FiPy can be found at http://www.ctcms.nist.gov/fipy/.

1.1 Download and Installation

Please refer to Chapter 2 for details on download and installation. FiPy can be redistributed and/or modified freely, provided that any derivative works bear some notice that they are derived from it, and any modified versions bear some notice that they have been modified.

1.2 Support

FiPy is being actively developed and supported. Please use the tracking system for bugs, support requests, feature requests and patch submissions. A mailing list is also available. We are also seeking collaborative efforts on this project.

1.3 Conventions and Notation

FiPy is driven by Python script files than you can view or modify in any text editor. FiPy sessions are invoked from a command-line shell, such as `tcsh` or `bash`.

Throughout, text to be typed at the keyboard will appear `like this`. Commands to be issued from an interactive shell will appear:

```
$ like this
```

where you would enter the text ("`like this`") following the shell prompt, denoted by "`$`".

Text blocks of the form:

```
>>> a = 3 * 4
>>> a
12
>>> if a == 12:
...     print "a is twelve"
...
a is twelve
```

are intended to indicate an interactive session in the Python interpreter. We will refer to these as "interactive sessions" or as "doctest blocks". The text "`>>>`" at the beginning of a line denotes the *primary prompt*, calling for input of a Python command. The text "`...`" denotes the *secondary prompt*, which calls for input that continues from the line above, when required by Python syntax. All remaining lines, which begin at the left margin, denote output from the Python interpreter. In all cases, the prompt is supplied by the Python interpreter and should not be typed by you.

Warning

Python is sensitive to indentation and care should be taken to enter text exactly as it appears in the examples.

When references are made to file system paths, it is assumed that the current working directory is the FiPy distribution directory, refered to as the "base directory", such that:

```
examples/diffusion/steadyState/mesh1D/input.py
```

will correspond to, *e.g.*:

```
/some/where/FiPy-0.1/examples/diffusion/steadyState/mesh1D/input.py
```

Paths will always be rendered using POSIX conventions. Any references of the form:

`examples.diffusion.steadyState.mesh1D.input`

are in the Python module notation and correspond to the equivalent POSIX path given above.

We may at times use a

> **Note**
>
> to indicate something that may be of interest

or a

> **Warning**
>
> to indicate something that could cause serious problems.

Chapter 2

Installation and Usage

The FiPy finite volume PDE solver relies on several third-party packages. It is *best to obtain and install those first*, before attempting to install FiPy.

Note

Most of the installation steps will involve a variant on the command:

```
$ python setup.py ...
```

In addition to the specific commands given here, further information about each `setup.py` script is available by typing:

```
$ python setup.py --help
```

2.1 Privileges

If you do not have administrative privileges on your computer, or if for any reason you don't want to tamper with your existing Python installation, most packages (including FiPy) will allow you to install to an alternate location. Instead of installing these packages with `python setup.py install`, you would use `python setup.py install --home=<dir>`, where `<dir>` is the desired installation directory (usually "~" to indicate your home directory). You will then need to append `<dir>/lib/python` to your `PYTHONPATH` environment variable. See the Alternate Installation section of the Python document "Installing Python Modules" [5] for more information, such as circumstances in which you should use `--prefix` instead of `--home`.

2.2 Prerequisites

2.2.1 Operating System

FiPy has been developed and tested on the Unix operating systems Mac OS X 10.3 and Debian Linux 3.0. We welcome reports of compatibility with other systems, along with any steps necessary to install.

The only elements of FiPy that are likely to be platform-dependent are the viewers. All other aspects should function on any platform that has a recent Python installation.

2.2.2 Required Packages

Warning

FiPy will not run if the following items are not installed.

Python

http://www.python.org/

FiPy is written in the Python language and requires a Python installation to run. Python comes pre-installed on many operating systems, which you can check by opening a terminal and typing `python`, *e.g.*:

```
$ python
Python 2.3 (#1, Sep 13 2003, 00:49:11)
...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If necessary, you can download and install it for your platform.

Note

FiPy requires at least version 2.3 of Python.

Numeric

http://www.numpy.org

Obtain and install the Numeric package. FiPy has been tested with version 23.1 of Numeric. The newer Numarray package is not supported at this time.

Note

Because of a peculiarity in the way that Numeric is structured, the `--home=<dir>` installation option described in Privileges will not work quite as intended. To correct this problem, add `<dir>/lib/python/Numeric` to your `PYTHONPATH` environment variable.

PySparse

FiPy requires a customized version of Roman Geus' PySparse package.

You can either download the PySparse archive or check it out via anonymous CVS download:

```
$ cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/fipy login
```

and press enter at the password prompt, then:

```
$ cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/fipy checkout pysparse
```

From within the `pysparse` base directory, follow its included instructions for building PySparse for your platform.

2.2.3 Recommended Packages

Note

These packages are not required to run FiPy, but they can be helpful.

SciPy

http://www.scipy.org/

Significantly improved performance has been achieved with the judicious use of C language inlining, via the weave module of the SciPy package.

In addition, a handful of test cases use functions from the SciPy library.

gmsh

http://www.geuz.org/gmsh/

It is possible to create irregular meshes with this package.

Warning

The Mac OS X distribution of gmsh provides a nice graphical tool, but unfortunately this tool cannot be used by FiPy. Please download the source distribution and build the `gmsh` command-line tool for your platform.

2.2.4 Viewers

FiPy will work perfectly well without them, but at least one of the following packages will be needed to allow viewing the results of FiPy calculations:

Pygist

http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/python/pygist.html

The Pygist package can be used to display simulation results. We have not succeeded in building the native Pygist viewer on Mac OS X and recommend building the package with the `--x11` option described in the documentation.

PyX

http://pyx.sourceforge.net/

PyX allows the production of publication quality graphics with TEX labels. If available, FiPy can use this package to view or print results.

Gnuplot-py

http://gnuplot-py.sourceforge.net

Gnuplot.py is a Python package that interfaces to gnuplot, the popular open-source plotting program.

2.3 Obtaining FiPy

FiPy is freely available for download via CVS or as a compressed archive. We recommend CVS over archives at this early stage of the development cycle. To obtain FiPy via anonymous CVS, issue the following commands:

```
$ cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/fipy login
```

and press enter at the password prompt, then:

```
$ cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/fipy checkout -r STABLE fipy
```

Further information about CVS can be found in Section 2.7 and at http://sourceforge.net/cvs/?group_id=118428.

2.3.1 Manual

You can download the latest manual or, if you have obtained FiPy via CVS, a fresh copy can be built by issuing the following command in the base directory:

```
$ python setup.py build_docs --latex --manual
```

The epydoc package and a command-line pdfTeX installation are required in order to build the manual.

2.4 Testing FiPy

From the base directory, you can verify that FiPy works properly by executing:

```
$ python setup.py test
```

Depending on the packages you chose to install in Recommended Packages, be sure to set the appropriate environment variables. You can expect a few errors if you did not install all of the recommended packages.

If you chose to install the weave package, you should rerun the tests with:

```
$ python setup.py test --inline
```

A few tests will fail the first time as a result of the messages output in the course of caching the compiled inline code, but a repeat test should have no failures.

2.5 Installing FiPy

Once you are confident that all of the requisite packages have been installed properly and FiPy passes its tests, you can install it by typing:

```
$ python setup.py install
```

at the command line. Alternatively, you may choose not to formally install FiPy and to simply work within the base directory instead.

Warning

Keep in mind that you will then need to preserve your changes when upgrades to FiPy become available (upgrades via CVS will handle this issue automatically).

If you wish to develop FiPy scripts outside of the distribution directory, but choose not to formally install FiPy, you will need to ensure that the FiPy distribution directory is appended to your `PYTHONPATH` environment variable.

2.6 Using FiPy

To see examples of problems that FiPy is capable of solving, you can run any of the scripts in Part II . All should have appropriate executable permissions, allowing you to type, *e.g.*:

```
$ examples/diffusion/steadyState/mesh1D/input.py
```

at the command line, which should produce a graphical display of the solution.

With judicious use of the weave package, we have been able to obtain significantly improved performance, while keeping the code as clear as possible. You can invoke this faster code by passing the `--inline` option at the command line, *i.e.*:

```
$ examples/diffusion/steadyState/mesh1D/input.py --inline
```

In order to customize the examples, or to develop your own scripts, some knowledge of Python syntax is required. We recommend you familiarize yourself with the excellent Python tutorial [6].

2.7 CVS tags

Most users will not want to download the latest state of FiPy in the CVS repository, as these files are subject to active development and may not behave as desired. Any released version of FiPy will be designated with a fixed tag:

`version-x_y` designates a released version x.y.

The current version of FiPy is 0.1.

Most users will not be interested in particular version numbers, but instead with the degree of code stability. Different "tracking tags" are used to indicate different stages of FiPy development. You will need to decide on your own risk tolerance when deciding which stage of development to track. The tracking tags applied FiPy, in decreasing order stability, are:

`STABLE` designates the most recent release in the repository that can be considered stable for daily use by the common user. This is a good tag to track if you don't want to run into bugs introduced with ongoing development but would like to take advantage of new features as soon as possible.

`CURRENT` designates the most recent code on the trunk forming a coherent state of FiPy, in general this will mean a release, but can also mean a pre-release testing version. For instance, the release engineer might ask testers to test `CURRENT` before he makes a release. This tracking tag is restricted to the trunk.

`HEAD` this is a CVS internal tag designating the latest revision of any file present in the repository. It is also a valid branch tag designating the trunk. For our purposes `HEAD` can be used as a tracking tag designating the very latest code checked into the repository; FiPy is not guaranteed to pass its tests or to be in a consistent state when checked out under this tag. This tracking tag is restricted to the trunk.

One final type of tracking tag to note:

`branch-version-x_y` designates a line of development based on a previously released version (i.e., if current development work is being spent on version 0.2, but a bug is found and fixed in version 0.1, that fix will be tagged as `version-0_1_1`, and can be obtained from `branch-version-0_1`).

Any other tags will not generally by of interest to most users.

A fresh copy of FiPy that is designated by a particular `<tag>` can be obtained with:

```
$ cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/fipy checkout -r <tag> fipy
```

An existing CVS checkout of FiPy can be shifted to a different state of development by issuing the command:

```
$ cvs update -r <tag>
```

Chapter 3

Theoretical and Numerical Background

This chapter describes the numerical methods used to solve equations in the FiPy programming environment. FiPy uses the finite volume method (FVM) to solve coupled sets of partial differential equations (PDEs). For a good introduction to the FVM see Nick Croft's PhD thesis [7], Patanker [8] or Versteek and Malalasekera [9].

Essentially, the FVM consists of dividing the solution domain into discrete finite volumes over which the state variables are approximated with linear or higher order interpolations. The derivatives in each term of the equation are satisfied with simple approximate interpolations in a process known as discretization. The (FVM) is a popular discretization technique employed to solve coupled PDEs used in many application areas (*e.g.* Fluid Dynamics).

3.1 General Conservation Equation

The equations that model the evolution of physical, chemical and biological systems often have a remarkably universal form. Indeed, PDEs have proven necessary to model complex physical systems and processes that involve variations in both space and time. In general, given a variable of interest ϕ such as species concentration, pH, or temperature, there exists an evolution equation of the form

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \tag{3.1}$$

where H is a function of ϕ, other state variables λ_i, and higher order derivatives of all of these variables. Examples of such systems are wide ranging, but include problems that exhibit a combination of diffusing and reacting species, as well as such diverse problems as determination of the electric potential in heart tissue, of fluid flow, stress evolution, and even the Schrödinger equation.

A general conservation equation, solved using FiPy, can include any combination of the following

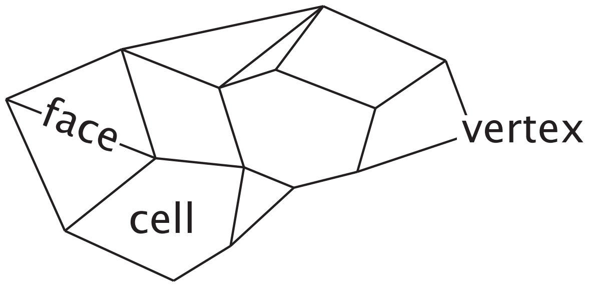

Figure 3.1: A mesh consists of cells, faces and vertices. For the purposes of FiPy, the divider between two cells is known as a face for all dimensions.

terms,

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} = \underbrace{\nabla\cdot(\vec{u}\phi)}_{\text{convection}} + \underbrace{\nabla\cdot(\Gamma_1\nabla\phi)}_{\text{diffusion}} + \underbrace{\left[\nabla\cdot(\Gamma_i\nabla)\right]^n\phi}_{n^{\text{th}}\text{ order}} + \underbrace{S_\phi}_{\text{source}} \tag{3.2}$$

where ρ, $\vec{u}$ and Γ_i represent coefficients in the transient, convection and diffusion terms respectively. These coefficients can be arbitrary functions of any parameters or variables in the system. The variable ϕ represents the unknown quantity in the equation. The n^{th} order term can represent any higher order diffusion-type term, where the order is given by the exponent n. For example, the n^{th} order term can represent a diffusion term when the exponent $n = 1$ or a Cahn-Hilliard term when $n = 2$. A Cahn-Hilliard term has the form [10, 11, 12],

$$\nabla\cdot\left(\Gamma_1\nabla\left[\nabla\cdot(\Gamma_2\nabla\phi)\right]\right) \tag{3.3}$$

Of course, higher order terms ($n > 2$) are also possible.

3.2 Finite Volume Method

To use the FVM, the solution domain must first be divided into non-overlapping polyhedral elements or cells. A solution domain divided in such a way is generally known as a mesh (as we will see, a `Mesh` is also a FiPy object). A mesh consists of vertices, faces and cells (see Figure 3.1). In the FVM the variables of interest are averaged over control volumes (CVs). The CVs are either defined by the cells or are centered on the vertices.

3.2.1 Cell Centered FVM (CC-FVM)

In the CC-FVM the CVs are formed by the mesh cells with the cell center 'storing' the average variable value in the CV, (see Figure 3.2). The face fluxes are approximated using the variable values in the two adjacent cells surrounding the face. This low order approximation has the advantage of being efficient and requiring matrices of low band width (the band width is equal to the number of cell neighbors plus one) and thus low storage requirement. However, the mesh topology is restricted

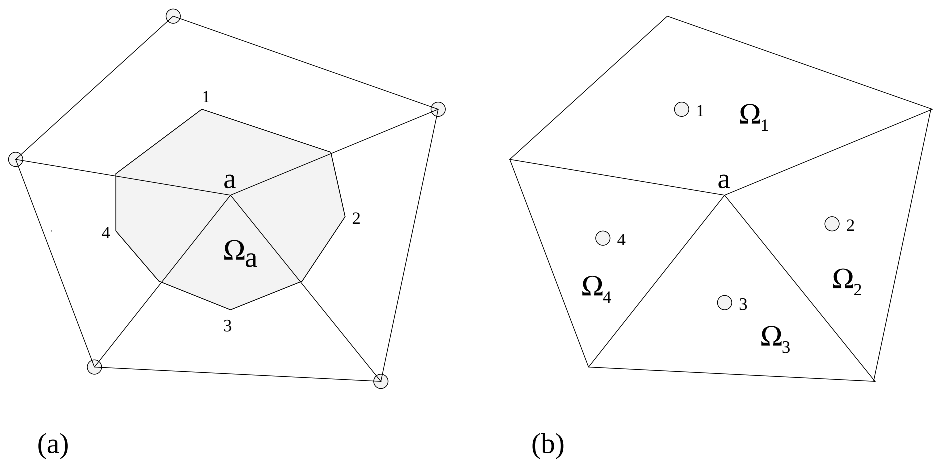

Figure 3.2: CV structure for an unstructured mesh, (a) Ω_a represents a vertex-based CV and (b) Ω_1, Ω_2, Ω_3 and Ω_4 represent cell centered CVs.

due to an orthogonality and conjunctionality requirement. The value at a face is assumed to be the average value over the face. On an unstructured mesh the face center may not lie on the line joining the CV centers, which will lead to an error in the face interpolation. FiPy currently only uses the CC-FVM.

3.2.2 Vertex Centered FVM (VC-FVM)

In the VC-FVM the CV is centered around the vertices and the cells are divided into sub-control volumes that make up the main CVs, (see Figure 3.2). The vertices 'store' the average variable values over the CVs. The CV faces are constructed within the cells rather than using the cell faces as in the CC-FVM. The face fluxes use all the vertex values from the cell where the face is located to calculate interpolations. For this reason, the VC-FVM is less efficient and requires more storage (a larger matrix band width) than the CC-FVM. However, the mesh topology does not have the same restrictions as the CC-FVM. Future releases of FiPy will have both the CC-FVM and VC-FVM capabilities.

3.3 Discretization

The first step in the discretization of Equation (3.2) using the CC-FVM is to integrate over a CV and then make appropriate approximations for fluxes across the boundary of each CV. In this section, each term in Equation (3.2) will be examined separately.

3.3.1 Transient Term

For the transient term, the discretization of the integral $\int_V$ over the volume of a CV is given by

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{\rho_P(\phi_P - \phi_P^{\text{old}})V_P}{\Delta t} \tag{3.4}$$

where ϕ_P represents the average value of ϕ in a CV centered on a point P and the superscript "old" represents the previous time-step value. The value V_P is the volume of the CV and Δt is the time step size.

3.3.2 Convection Term

The discretization for the convection term is given by

$$\begin{aligned} \int_V \nabla \cdot (\vec{u}\phi)\, dV &= \int_S (\vec{n} \cdot \vec{u})\phi\, dS & (3.5) \\ &\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f & (3.6) \end{aligned}$$

where we have used the divergence theorem to transform the integral over the CV volume $\int_V$ into an integral over the CV surface $\int_S$. The summation over the faces of a CV is denoted by $\sum_f$ and A_f is the area of each face. The vector $\vec{n}$ is the normal to the face pointing out of the CV into an adjacent CV centered on point A. When using a first order approximation, the value of ϕ_f must depend on the average value in adjacent cell ϕ_A and the average value in the cell of interest ϕ_P, such that

$$\phi_f = \alpha_f \phi_P + (1 - \alpha_f)\phi_A. \tag{3.7}$$

The weighting factor α_f is determined by the convection scheme, described later in this chapter.

3.3.3 Diffusion Term

The discretization for the diffusion term is given by

$$\begin{aligned} \int_V \nabla \cdot (\Gamma \nabla \phi) dV &= \int_S \Gamma(\vec{n} \cdot \nabla\phi) dS & (3.8) \\ &\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla\phi)_f A_f & (3.9) \end{aligned}$$

The estimation for the flux, $(\vec{n} \cdot \nabla\phi)_f$, is obtained via

$$(\vec{n} \cdot \nabla\phi)_f \simeq \frac{\phi_A - \phi_P}{d_{AP}} \tag{3.10}$$

where the value of d_{AP} is the distance between neighboring cell centers. This estimate relies on the orthogonality of the mesh, and becomes increasingly inaccurate as the non-orthogonality increases. Correction terms have been derived to improve this error but are not currently included in FiPy [7].

3.3.4 Source Term

The discretization for the source term is given by,

$$\int_V S_\phi \, dV \simeq S_\phi V_P. \tag{3.11}$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (3.11) becomes

$$V_P(S_0 - S_1 \phi_P), \tag{3.12}$$

where S_0 is the source which is independent of ϕ and S_1 is the coeficient of the source which is linearly dependent on ϕ.

3.4 Linear Equations

The aim of the discretization is to reduce the continuous general equation to a set of discrete linear equations that can then be solved to obtain the value of the dependent variable at each CV center. This results in a sparse linear system that requires an efficient iterative scheme to solve. The iterative schemes available to FiPy are currently encapsulated in the spmatrix suite of solvers and include most common solvers such as the conjugate gradient method and LU decomposition. There are plans to include other solver suites that are compatible with Python.

Combining Equations (3.4), (3.6), (3.9) and (3.11), the complete discretization for equation (3.2) can now be written for each CV as

$$\begin{aligned} \frac{\rho_P(\phi_P - \phi_P^{\text{old}})V_P}{\Delta t} &= \sum_f (\vec{n} \cdot \vec{u})_f A_f \left[\alpha_f \phi_P + (1 - \alpha_f)\, \phi_A\right] \\ &\quad + \sum_f \Gamma_f A_f \frac{(\phi_A - \phi_P)}{d_{AP}} + V_P(S_0 - S_1 \phi_P). \end{aligned} \tag{3.13}$$

Equation (3.13) is now in the form of a set of linear combinations between each CV value and its neighboring values and can be written in the form

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \tag{3.14}$$

where

$$a_P = V_P S_1 + \frac{\rho_P V_P}{\Delta t} + \sum_f (a_A - F_f), \tag{3.15}$$

$$a_A = (1 - \alpha_f) F_f + D_f, \tag{3.16}$$

$$b_P = V_P S_0 + \frac{\rho_P V_P \phi_P^{\text{old}}}{\Delta t}. \tag{3.17}$$

The face coefficients, F_f and D_f, represent the convective strength and diffusive conductance respectively, and are given by

$$\begin{aligned} F_f &= A_f(\vec{u} \cdot \vec{n})_f, && (3.18) \\ D_f &= \frac{A_f \Gamma_f}{d_{AP}}. && (3.19) \end{aligned}$$

3.5 Numerical Schemes

The coefficients of equation (3.14) must remain positive, since an increase in a neighboring value must result in an increase in ϕ_P to obtain physically realistic solutions. Thus, the inequalities $a_A > 0$ and $a_A - F_f > 0$ must be satisfied. The Peclet number $P_f \equiv -F_f/D_f$ is the ratio between convective strength and diffusive conductance. To achieve physically realistic solutions, the inequality

$$\frac{1}{1-\alpha_f} > P_f > -\frac{1}{\alpha_f} \tag{3.20}$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (3.20). The various differencing schemes are:

the central differencing scheme, where

$$\alpha_f = \frac{d_{Af}}{d_{Af} + d_{fP}}. \tag{3.21}$$

In many circumstances with a structured mesh, $\alpha_f = 1/2$, so that $|P_f| < 2$ satisfies Equation (3.20). Thus, the central differencing scheme is only numerically stable for a low values of P_f.

the upwind scheme, where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \tag{3.22}$$

Equation (3.22) satisfies the inequality in Equation (3.20) for all values of P_f. However the solution over predicts the diffusive term leading to excessive numerical smearing ("false diffusion").

the exponential scheme, where

$$\alpha_f = \frac{(P_f - 1)\exp(P_f) + 1}{P_f(\exp(P_f) - 1)}. \tag{3.23}$$

This formulation can be derived from the exact solution, and thus, guarantees positive coefficients while not over-predicting the diffusive terms. However, the computation of exponentials is slow and therefore a faster scheme is generally used, especially in higher dimensions.

the hybrid scheme, where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \tag{3.24}$$

The hybrid scheme is formulated by allowing $P_f \to \infty$, $P_f \to 0$ and $P_f \to -\infty$ in the exponential scheme. The hybrid scheme is an improvement on the upwind scheme, however, it deviates from the exponential scheme at $|P_f| = 2$.

the power law scheme, where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \tag{3.25}$$

The power law scheme overcomes the inaccuracies of the hybrid scheme, while improving on the computational time for the exponential scheme.

All of the numerical schemes presented here are available in FiPy and can be selected by the user.

Chapter 4

Design and Implementation

The goal of FiPy is to provide a highly customizable, open source code for modeling problems involving coupled sets of PDEs. FiPy allows users to select and customize modules from within the framework. The initial implementation of FiPy has been developed to address model problems in materials science such as poly-crystals, dendritic growth and electrochemical deposition. These applications all contain various combinations of PDEs with differing forms in conjunction with other unusual physics (over varying length scales) and unique solution procedures. The philosophy of FiPy is to enable customization while providing a library of efficient modules for common objects and data types.

4.1 Design

4.1.1 Numerical Approach

The solution algorithms given in the FiPy examples involve combining sets of PDEs while tracking an interface where the parameters of the problem change rapidly. The phase field method and the level set method are specialized techniques to handle the solution of PDEs in conjunction with a deforming interface. FiPy contains several examples of both methods.

FiPy uses the well-known Finite Volume Method (FVM) to reduce the model equations to a form tractable to linear solvers.

4.1.2 Object Oriented Structure

FiPy is programmed in an object-oriented manner. The benefit of object oriented programming mainly lies in encapsulation and inheritance. Encapsulation refers to the tight integration between certain pieces of data and methods that act on that data. Encapsulation allows parts of the code to be separated into clearly defined independent modules that can be re-applied or extended in new ways. Inheritance allows code to be reused, overridden, and new capabilities to be added without altering the original code. An object is treated by its users as an abstraction; the details of its implementation and behavior are internal.

4.1.3 Test Based Development

FiPy has been developed with a large number of test cases. These test cases are in two categories. The lower level tests operate on the core modules at the individual method level. The aim is that every method within the core installation has a test case. The high level test cases operate in conjunction with example solutions and serve to test global solution algorithms and the interaction of various modules.

With this two-tiered battery of tests, at any stage in code development, the test cases can be executed and errors can be identified. A comprehensive test base provides reassurance that any code breakages will be clearly demonstrated with a broken test case. A test base also aids dissemination of the code by providing simple examples and knowledge of whether the code is working on a particular computer environment.

4.1.4 Open Source

In recent years, there has been a movement to release software under open source and associated unrestrictied licenses, especially within the scientific community. These licensing terms allow users to develop their own applications with complete access to the source code and then either contribute back to the main source repository or freely distribute their new adapted version.

As a product of the National Institute of Standards and Technology, the FiPy framework is placed in the public domain as a matter of U. S. Federal law. Furthermore, FiPy is built upon existing open source tools. Others are free to use FiPy as they see fit and we welcome contributions to make FiPy better.

4.1.5 High-Level Scripting Language

Programming languages can be broadly lumped into two categories: compiled languages and interpreted (or scripting) languages. Compiled languages are converted from a human-readable text source file to a machine-readable binary application file by a sequence of operations generally referred to as "compiling" and "linking". The binary application can then be run as many times as desired, but changes will provoke a new cycle of compiling and linking. Interpreted languages are converted from human-readable to machine-readable on the fly, each time the script is executed. Because the conversion happens every time[1], interpreted code is usually slower when running than compiled code. On the other hand, code development and debugging tends to be much easier and fluid when it's not necessary to wait for compile and link cycles after every change. Furthermore, because the conversion happens in real time, it is possible to have interactive sessions in a scripting language that are not generally possible in compiled languages.

[1]. . . neglecting such common optimizations as byte-code interpreters

Another distinction, somewhat orthogonal, but closely related, to that between compiled and interpreted languages, is between low-level languages and high-level languages. Low-level languages describe actions in simple terms that are closer to the way the computer actually functions. High-level languages describe actions in more complex and abstract terms that are closer to the way the programmer thinks about the problem at hand. This increased complexity in the meaning of an expression renders simpler code, because the details of the implementation are hidden away in the language internals or in an external library. For example, a low-level matrix multiplication written in C might be rendered as

```
if (Acols != Brows)
    error "these matrix shapes cannot be multiplied";

C = (float *) malloc(sizeof(float) * Bcols * Arows);

for (i = 0; i < Bcols; i++) {
    for (j = 0; j < Arows; j++) {
        C[i][j] = 0;
        for (k = 0; k < Acols; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Note that the dimensions of the arrays must be supplied externally, as C provides no intrinsic mechanism for determining the shape of an array. An equivalent high-level construction might be as simple as

```
C = A * B
```

All of the error checking, dimension measuring, and space allocation is handled automatically by low-level code that is intrinsic to the high-level matrix multiplication operator. The high-level code "knows" that matrices are involved, how to get their shapes, and to interpret '`*`' as a matrix multiplier instead of an arithmetic one. All of this allows the programmer to think about the operation of interest and not worry about introducing bugs in low-level code that is not unique to their application.

Although it needn't be true, for a variety of reasons, compiled languages tend to be low-level and interpreted languages tend to be high-level. Because low-level languages operate closer to the intrinsic "machine language" of the computer, they tend to be faster at running a given task than high-level languages, but programs written in them take longer to write and debug. Because running performance is a paramount concern, most scientific codes are written in low-level compiled languages like FORTRAN or C.

A rather common scenario in the development of scientific codes is that the first draft hard-codes all of the problem parameters. After a few (hundred) iterations of recompiling and relinking the application to explore changes to the parameters, code is added to read an input file containing a list of numbers. Eventually, the point is reached where it is impossible to remember which parameter comes in which order or what physical units are required, so code is added to, for example, interpret a line beginning with '`#`' as a comment. At this point, the scientist has begun developing a scripting language without even knowing it. Unfortunately for them, very few scientists have actually studied computer science or actually know anything about the design and implementation of script interpreters. Even if they have the expertise, the time spent developing such a language interpreter is time not spent actually doing research.

In contrast, a number of very powerful scripting languages, such as Tcl, Java, Python, Ruby, and even the venerable BASIC, have open source interpreters that can be embedded directly in an application, giving scientific codes immediate access to a high-level scripting language designed by someone who actually knew what they were doing.

We have chosen to go a step further and not just embed a full-fledged scripting language in the FiPy framework, but instead to design the framework from the ground up in a scripting language. While runtime performance is unquestionably important, many scientific codes are run relatively little, in proportion to the time spent developing them. If a code can be developed in a day instead of a month, it may not matter if it takes another day to run instead of an hour. Furthermore, there are a variety of mechanisms for diagnosing and optimizing those portions of a code that are actually time-critical, rather than attempting to optimize all of it by using a language that is more palatable to the computer than to the programmer. Thus FiPy, rather than taking the approach of writing the fast numerical code first and then dealing with the issue of user interaction, initially implements most modules in high-level scripting language and only translates to low-level compiled code those portions that prove inefficient.

4.1.6 Python Programming Language

Acknowledging that several scripting languages offer a number, if not all, of the features described above, we have selected Python for the implementation of FiPy. Python is:

- an interpreted language that combines remarkable power with very clear syntax,
- freely usable and distributable, even for commercial use,
- fully object oriented,
- distributed with powerful automated testing tools (doctest, unittest),
- actively used and extended by other scientists and mathemeticians (SciPy, Numeric, Scientific Python, PySparse).
- easily integrated with low-level languages such as C (weave, blitz, PyRex).

4.2 Implementation

The Python classes that make up FiPy are described in detail in the *FiPy Programmer's Reference*, but we give a brief overview here. FiPy is based around three fundamental Python classes: `Mesh`, `Variable`, and `Equation`. Using the terminology of Chapter 3:

A `Mesh` object represents the domain of interest. FiPy contains many different specific mesh classes to describe different geometries.

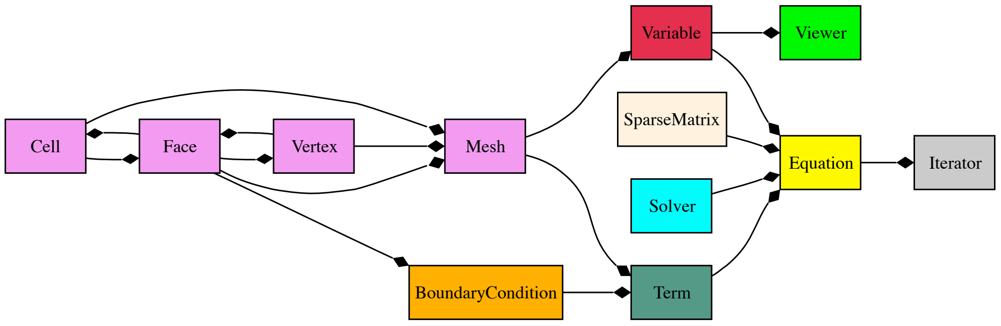

Figure 4.1: Primary object relationships in FiPy.

A `Variable` object represents a quantity or field that can change during the problem evolution. A particular type of `Variable`, called a `CellVariable`, represents ϕ at the centers of the `Cell`s of the `Mesh`. A `CellVariable` describes the values of the field ϕ, but it is not concerned with their geometry; that role is taken by the `Mesh`.

An important property of `Variable` objects is that they can describe dependency relationships, such that:

```
>>> a = Variable(value = 3)
>>> b = a * 4
```

does not assign the value `12` to `b`, but rather it assigns a multiplication operator object to `b`, which depends on the `Variable` object `a`:

```
>>> b
(Variable(value = 3) * 4)
>>> a.setValue(5)
>>> b
(Variable(value = 5) * 4)
```

The numerical value of the `Variable` is not calculated until it is needed (a process known as "lazy evaluation"):

```
>>> print b
20
```

An `Equation` object represents Equation (3.1).

Beyond these three fundamental classes, FiPy is composed of a number of related classes. The relationships between these classes are shown in Figure 4.1. A `Mesh` object is composed of `Cell` objects. Each `Cell` is defined by its bounding `Face` objects and each `Face` is defined by its bounding `Vertex` objects. As shown conceptually in Equation (3.2), an `Equation` is composed of multiple `Term` objects, which encapsulate the building of different parts of the `SparseMatrix` that defines the solution of the `Equation`. `BoundaryCondition` objects are used to describe the conditions on the boundaries of the `Mesh`, and each `Term` interprets the `BoundaryCondition` objects as necessary to modify the `SparseMatrix`. An `Iterator` object handles the iterative solution of a set of `Equation` objects until some desired convergence criterion has been met. Each `Equation` can apply a unique `Solver` to invert its `SparseMatrix` in the most expedient and stable fashion. At any point during the solution, a `Viewer` can be invoked to display the values of the solved `Variable` objects.

At this point, it will be useful to examine some of the example problems in Part II. More classes are introduced in the examples, along with illustrations of their instantiation and use.

Part II

Examples

Note

Any given "Module example.something.input" can be found in the file "`examples/something/input.py`".

These examples can be used in at least four ways:

- Each example can be invoked individually to demonstrate an application of FiPy:

  ```
  $ examples/something/input.py
  ```

- Each example can be invoked such that when it has finished running, you will be left in an interactive Python shell:

  ```
  $ python -i examples/something/input.py
  ```

  At this point, you can enter Python commands to manipulate the model or to make queries about the example's variable values. For instance, the interactive Python sessions in the example documentation can be typed in directly to see that the expected results are obtained.

- Alternatively, these interactive Python sessions, known as doctest blocks, can be invoked as automatic tests:

  ```
  $ python setup.py test --examples
  ```

  In this way, the documentation and the code are always certain to be consistent.

- Finally, and most importantly, the examples can be used as a templates to design your own problem scripts.

  Note

  The examples shown in this manual have been written with particular emphasis on serving as both documentation and as comprehensive tests of the FiPy framework. As explained at the end of `examples/diffusion/steadyState/mesh1D.py`, your own scripts can be much more succint, if you wish, and include only the text that follows the ">>>" and "`...`" prompts shown in these examples.
  To obtain a copy of an example, containing just the script instructions, type:

  ```
  $ python setup.py copy_script --From x.py --To y.py
  ```

In addition to those presented in this manual, there are dozens of other files in the `examples/` directory (all with "`input`" in their title), that demonstrate other uses of FiPy. If these examples do not help you construct your own problem scripts, please contact us.

Example Contents

Chapter 5

Diffusion Examples

5.1 Module examples.diffusion.steadyState.mesh1D.input

To run this example from the base FiPy directory, type:

```
$ examples/diffusion/steadyState/mesh1D/input.py
```

at the command line. A display of the result should appear and the word `finished` in the terminal.

This example takes the user through assembling a simple problem with FiPy. It describes a steady 1D diffusion problem with fixed value boundary conditions such that,

$$\nabla \cdot (D \nabla \phi) = 0$$

with initial conditions $\phi = 0$ at $t = 0$, boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = 1, \end{cases}$$

and parameter value $D = 1$. The first step is to create a mesh with 50 elements. The `Grid2D` object represents a rectangular structured grid. The parameters `dx` and `dy` refer to the grid spacing (set to unity here).

```
>>> nx = 50
>>> ny = 1
>>> dx = 1.
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

The solution of all equations in FiPy requires a variable. These variables store values on various parts of the mesh. In this case we need a `CellVariable` object as the solution is sought on the cell centers. The boundary conditions are given by `valueLeft = 0` and `valueRight = 1`. The initial value for the variable is set to `value = valueLeft`.

```
>>> valueLeft = 0
>>> valueRight = 1
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "solution variable", mesh = mesh, value = valueLeft)
```

Boundary conditions are given to the equation via a `Tuple` (list). Boundary conditions are formed with a value and a set of faces over which they apply. For example here the exterior faces on the left of the domain are extracted by `mesh.getFacesLeft()`. These faces and a value (`valueLeft`) are passed to a `FixedValue` boundary condition. A fixed flux of zero is set on the top and bottom surfaces to simulate a one dimensional problem. The `FixedFlux(someFaces, 0.)` is the default boundary condition if no boundary conditions are specified for exterior faces.

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (FixedFlux(mesh.getFacesTop(),0.),
...                       FixedFlux(mesh.getFacesBottom(),0.),
...                       FixedValue(mesh.getFacesRight(),valueRight),
...                       FixedValue(mesh.getFacesLeft(),valueLeft))
```

A solver is created and passed to the equation. This solver uses an iterative conjugate gradient method to solve implicitly at each time step.

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
>>> solver = LinearPCGSolver(tolerance = 1.e-15, steps = 1000)
```

An equation is passed coefficient values, boundary conditions and a solver. The equation knows how to assemble and solve a system matrix. The `DiffusionEquation` object in FiPy represents a general transient diffusion equation

$$\frac{\partial(\tau\phi)}{\partial t} = \nabla \cdot (D\nabla\phi).$$

We solve this equation in steady state by setting $\tau = 0$, which is accomplished by setting the `transientCoeff` parameter to 0:

```
>>> from fipy.equations.diffusionEquation import DiffusionEquation
>>> eq = DiffusionEquation(var,
...                        transientCoeff = 0.,
...                        diffusionCoeff = 1.,
...                        solver = solver,
...                        boundaryConditions = boundaryConditions)
```

The `Iterator` object takes a `Tuple` of equations and solves to a required tolerance for the given equations at each time step.

```
>>> from fipy.iterators.iterator import Iterator
>>> iterator = Iterator((eq,))
```

Here the iterator does one time step to implicitly find the steady state solution.

```
>>> iterator.timestep()
```

To test the solution, the analytical result is required. The `x` coordinates from the mesh are gathered and the length of the domain `Lx` is calculated. An array, `analyticalArray`, is calculated to compare with the numerical result,

```
>>> x = mesh.getCellCenters()[:,0]
>>> Lx = nx * dx
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
```

Finally the analytical and numerical results are compared with a tolerance of `1e-10`.

```
>>> import Numeric
>>> Numeric.allclose(var, analyticalArray, rtol = 1e-10, atol = 1e-10)
1
```

A `Viewer` object allows a variable to be displayed. Here we are using the Gist package to view the field. The Gist viewer is constructed and the results are viewed:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var, minVal =0., maxVal = 1.)
...     viewer.plot()
```

If this example had been written primarily as a script, instead of as documentation, we would delete every line that does not begin with either “`>>>`” or “`...`”, and then delete those prefixes from the remaining lines, leaving:

```
nx = 50
ny = 1
dx = 1.
dy = 1.
from fipy.meshes.grid2D import Grid2D
mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)

valueLeft = 0
valueRight = 1
from fipy.variables.cellVariable import CellVariable
var = CellVariable(name = "solution variable", mesh = mesh, value = valueLeft)

from fipy.boundaryConditions.fixedValue import FixedValue
from fipy.boundaryConditions.fixedFlux import FixedFlux
boundaryConditions = (FixedFlux(mesh.getFacesTop(),0.),
                      FixedFlux(mesh.getFacesBottom(),0.),
                      FixedValue(mesh.getFacesRight(),valueRight),
                      FixedValue(mesh.getFacesLeft(),valueLeft))

from fipy.solvers.linearPCGSolver import LinearPCGSolver
solver = LinearPCGSolver(tolerance = 1.e-15, steps = 1000)
```

```
from fipy.equations.diffusionEquation import DiffusionEquation
eq = DiffusionEquation(var,
                       transientCoeff = 0.,
                       diffusionCoeff = 1.,
                       solver = solver,
                       boundaryConditions = boundaryConditions)

from fipy.iterators.iterator import Iterator
iterator = Iterator((eq,))

iterator.timestep()

x = mesh.getCellCenters()[:,0]
Lx = nx * dx
analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx

import Numeric
Numeric.allclose(var, analyticalArray, rtol = 1e-10, atol = 1e-10)

from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
viewer = Grid2DGistViewer(var, minVal =0., maxVal = 1.)

if __name__ == '__main__':
    viewer.plot()
```

Your own scripts will tend to look like this, although you can always write them as doctest scripts if you choose. You can obtain a plain script like this from one of the examples by typing:

```
$ python setup.py copy_script --From examples/.../input.py --To myInput.py
```

at the command line.

Most of the FiPy examples will be a mixture of plain scripts and doctest documentation/tests.

5.2 Module examples.diffusion.steadyState.mesh20x20.input

This input file again solves a steady 1D diffusion problem as in `examples/diffusion/steadyState/mesh1D/input.py`, the difference being that the mesh is two dimensional:

```
>>> nx = 20
>>> ny = 20
>>> dx = 1.
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

We create a `CellVariable` and initialize it to `valueLeft`:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "solution variable",
...                    mesh = mesh,
...                    value = valueLeft)
```

We create a diffusion equation, which is solved with an iterative conjugate gradient solver. We apply Dirichlet boundary conditions to the left and right and Neumann boundary conditions to the top and bottom.

```
>>> from fipy.equations.diffusionEquation import DiffusionEquation
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> eq = DiffusionEquation(var,
...                        transientCoeff = 0.,
...                        diffusionCoeff = 1.,
...                        solver = LinearPCGSolver(tolerance = 1.e-15,
...                                                 steps = 1000
...                                                 ),
...                        boundaryConditions = (FixedValue(mesh.getFacesLeft(),valueLeft),
...                                              FixedValue(mesh.getFacesRight(),valueRight),
...                                              FixedFlux(mesh.getFacesTop(),0.),
...                                              FixedFlux(mesh.getFacesBottom(),0.)
...                                              )
...                        )
```

We iterate the diffusion equation to equilibrium:

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> it.timestep()
```

The result is again tested against the expected linear composition profile:

```
>>> Lx = nx * dx
>>> x = mesh.getCellCenters()[:,0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> import Numeric
>>> Numeric.allclose(var, analyticalArray, rtol = 1e-10, atol = 1e-10)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

5.3 Module examples.diffusion.explicit.mesh10.input

This input file again solves a 1D diffusion problem as in `examples/diffusion/steadyState/mesh1D/input.py`, the difference being that this transient example is solved explicitly.

We create a 1D mesh:

```
>>> nx = 100
>>> ny = 1
>>> dx = 1.
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx, dy, nx, ny)
```

and we initialize a `CellVariable` to `initialValue`:

```
>>> valueLeft = 0.
>>> initialValue = 1.
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = initialValue)
```

The transient equation

$$\frac{\partial(\tau\phi)}{\partial t} = \nabla \cdot (D \nabla \phi)$$

is represented by the `ExplicitDiffusionEquation`, which includes a `TransientTerm`. The coefficient of the `TransientTerm` depends on the desired time step.

```
>>> timeStepDuration = 0.1
```

We take the diffusion coefficient $D = 1$

```
>>> diffusionCoeff = 1.
```

We build the equation with an appropriate solver and boundary conditions:

```
>>> from fipy.equations.explicitDiffusionEquation import ExplicitDiffusionEquation
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> eq = ExplicitDiffusionEquation(var,
...                                transientCoeff = 1. / timeStepDuration,
...                                diffusionCoeff = diffusionCoeff,
...                                solver = LinearPCGSolver(tolerance = 1.e-15,
...                                                         steps = 1000
...                                ),
...                                boundaryConditions=(
...                                    FixedValue(mesh.getFacesLeft(),valueLeft),
...                                          FixedFlux(mesh.getFacesRight(),0),
...                                          FixedFlux(mesh.getFacesTop(),0.),
...                                          FixedFlux(mesh.getFacesBottom(),0.)
...                                     )
... )
```

In this case, many steps have to be taken to reach equilibrium. A loop is required to execute the necessary time steps:

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> steps = 100
>>> for step in range(steps):
...     it.timestep()
```

The analytical solution for this transient diffusion problem is given by $\phi = \operatorname{erf}(x/2\sqrt{Dt})$. The result is tested against the expected profile:

```
>>> Lx = nx * dx
>>> x = mesh.getCellCenters()[:,0]
>>> t = timeStepDuration * steps
>>> import Numeric
>>> epsi = x / Numeric.sqrt(t * diffusionCoeff)
>>> import scipy
>>> analyticalArray = scipy.special.erf(epsi/2)
>>> Numeric.allclose(var, analyticalArray, atol = 2e-3)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

5.4 Module examples.diffusion.variable.mesh10x1.input

This example is a 1D steady state diffusion test case with a diffusion coefficient that spatially varies such that

$$\frac{\partial}{\partial x} D \frac{\partial \phi}{\partial x} = 0,$$

with boundary conditions $\phi = 0$ at $x = 0$ and $D\frac{\partial \phi}{\partial x} = 1$ at $x = L$. The diffusion coefficient varies with the profile

$$D = \begin{cases} 1 & \text{for } 0 < x < L/4, \\ 0.1 & \text{for } L/4 \le x < 3L/4, \\ 1 & \text{for } 3L/4 \le x < L, \end{cases}$$

where

```
>>> L = 1.
```

is the length of the bar. Accurate answers to this problem are given for any number of cells where `nCells = 4 * i + 2` where `i` is an integer and of course for large `nCells`. In this example

```
>>> nx = 10
```

We create a 1D mesh of the appropriate size

```
>>> ny = 1
>>> dx = L / nx
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx, dy, nx, ny)
```

and initialize the solution variable to `valueLeft`:

```
>>> valueLeft = 0.
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "solution variable",
...     mesh = mesh,
...     value = valueLeft)
```

In this example, the diffusion coefficient is a numerical array that is passed to the diffusion equation. The diffusion coefficient exists on the faces of the cells and thus has to be the length of the faces. It is created in the following way:

```
>>> x = mesh.getFaceCenters()[:,0]
>>> import Numeric
>>> outerFaces = Numeric.logical_or(x < L / 4., x >= 3. * L / 4.)
>>> diffCoeff = Numeric.where(outerFaces, 1., 0.1)
```

We seek a steady-state solution, so the `transientCoeff` of the `DiffusionEquation` is set to zero. For boundary conditions, we have no-flux conditions top and bottom, a fixed value of `valueLeft` to the left, and a fixed flux of

```
>>> fluxRight = 1.
```

to the right:

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
>>> from fipy.equations.diffusionEquation import DiffusionEquation
>>> eq = DiffusionEquation(
...     var,
...     transientCoeff = 0.,
...     diffusionCoeff = diffCoeff,
...     solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 1000
...     ),
...     boundaryConditions=(
...         FixedValue(mesh.getFacesLeft(),valueLeft),
...         FixedFlux(mesh.getFacesRight(),fluxRight),
...         FixedFlux(mesh.getFacesTop(),0.),
...         FixedFlux(mesh.getFacesBottom(),0.)
...     )
... )
```

We iterate one time step to implicitly find the steady state solution:

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> it.timestep()
```

A simple analytical answer can be used to test the result:

$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \leq x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \leq x < L, \end{cases}$$

or

```
>>> x = mesh.getCellCenters()[:,0]
>>> values = x + 18. * L / 4.
>>> values = Numeric.where(x < 3. * L / 4., 10 * x - 9. * L / 4., values)
>>> values = Numeric.where(x < L / 4., x, values)
>>> Numeric.allclose(values, var, atol = 1e-8, rtol = 1e-8)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var, maxVal = L + 18. * L / 4.)
...     viewer.plot()
```

5.5 Module examples.diffusion.nthOrder.input2ndOrder1D

In this problem, we demonstrate the use of the `NthOrderDiffusionEquation` class in the simple case of steady state 1D diffusion, which was introduced in `examples/diffusion/steadyState/mesh1D/input.py`, to solve

$$\nabla \cdot (D \nabla \phi) = 0.$$

This examples shows that the `NthOrderDiffusionEquation` is equivalent to the `DiffusionEquation` when $n = 2$.

We create an appropriate 1D mesh:

```
>>> nx = 10
>>> ny = 1
>>> dx = 1.
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

and initialize the solution variable to `valueLeft`:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                    mesh = mesh,
...                    value = valueLeft)
```

The order n of the `NthOrderDiffusionEquation` is determined by twice the number of diffusion coefficients it is created with, so a single diffusion coefficient `(1.,)` gives $n = 2$. The diffusion equation is again solved with an iterative conjugate gradient solver. We apply Dirichlet boundary conditions to the left and right and Neumann boundary conditions to the top and bottom.

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> from fipy.equations.nthOrderDiffusionEquation import NthOrderDiffusionEquation
>>> eq = NthOrderDiffusionEquation(
...     var,
...     transientCoeff = 0.,
...     diffusionCoeff = (1.,),
...     solver = LinearPCGSolver(tolerance = 1.e-15,
...                              steps = 1000
...     ),
...     boundaryConditions = (FixedValue(mesh.getFacesLeft(),valueLeft),
...                           FixedValue(mesh.getFacesRight(),valueRight),
...                           FixedFlux(mesh.getFacesTop(),0.),
...                           FixedFlux(mesh.getFacesBottom(),0.)
...                           )
... )
```

We iterate one timestep to equilibrium:

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> it.timestep()
```

The result is tested against the expected linear diffusion profile:

```
>>> Lx = nx * dx
>>> x = mesh.getCellCenters()[:,0]
>>> import Numeric
>>> value = valueLeft + (valueRight - valueLeft) * x / Lx
>>> Numeric.allclose(var, value, rtol = 1e-10, atol = 1e-10)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

5.6 Module examples.diffusion.nthOrder.input4thOrder1D

This example uses the `NthOrderBoundaryCondition` class to solve the equation

$$\frac{\partial^4 \phi}{\partial x^4} = 0$$

on a 1D mesh of length

```
>>> L = 1000.
```

We create an appropriate mesh

```
>>> nx = 1000
>>> ny = 1
>>> dx = L / nx
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx, dy, nx, ny)
```

and initialize the solution variable to 0

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = 0.)
```

For this problem, we impose the boundary conditions:

$$\begin{aligned} \phi &= \alpha_1 \quad \text{at } x = 0 \\ \frac{\partial \phi}{\partial x} &= \alpha_2 \quad \text{at } x = L \\ \frac{\partial^2 \phi}{\partial x^2} &= \alpha_3 \quad \text{at } x = 0 \\ \frac{\partial^3 \phi}{\partial x^3} &= \alpha_4 \quad \text{at } x = L. \end{aligned}$$

or

```
>>> alpha1 = 2.
>>> alpha2 = 1.
>>> alpha3 = 4.
>>> alpha4 = -3.
```

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> from fipy.boundaryConditions.nthOrderBoundaryCondition \
...     import NthOrderBoundaryCondition
>>> boundaryConditions=(
...     FixedValue(mesh.getFacesLeft(), alpha1),
...     FixedFlux(mesh.getFacesRight(), alpha2),
...     NthOrderBoundaryCondition(mesh.getFacesLeft(), alpha3, 2),
...     NthOrderBoundaryCondition(mesh.getFacesRight(), alpha4, 3))
```

We initialize the steady-state equation and use the `LinearLUSolver` for stability.

```
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
```

By assigning two diffusion coefficients

```
>>> diffusionCoeff = (-1., 1.)
```

we obtain a fourth-order diffusion equation

```
>>> from fipy.equations.nthOrderDiffusionEquation import NthOrderDiffusionEquation
>>> eq = NthOrderDiffusionEquation(
...         var,
...         transientCoeff = 0.0,
...         diffusionCoeff = diffusionCoeff,
...         solver = LinearLUSolver(tolerance = 1e-11),
...         boundaryConditions=boundaryConditions)
```

We perform one implicit timestep to achieve steady state

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> it.timestep()
```

The analytical solution is:

$$\phi = \frac{\alpha_4}{6}x^3 + \frac{\alpha_3}{2}x^2 + \left(\alpha_2 - \frac{\alpha_4}{2}L^2 - \alpha_3 L\right)x + \alpha_1$$

or

```
>>> x = mesh.getCellCenters()[:,0]
>>> answer = alpha4 / 6. * x**3 + alpha3 / 2. * x**2
>>> answer += (alpha2 - alpha4 / 2. * L**2 - alpha3 * L) * x + alpha1
>>> import Numeric
>>> Numeric.allclose(answer, var, atol = 1e-10)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

Chapter 6

Convection Examples

6.1 Module examples.convection.exponential1D.input

This example solves the steady-state convection-diffusion equation given by:

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10, 0)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,0.)
```

We define a 1D mesh

```
>>> L = 10.
>>> nx = 1000
>>> ny = 1
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight),
...     FixedFlux(mesh.getFacesTop(), 0.),
...     FixedFlux(mesh.getFacesBottom(), 0.)
...     )
```

The solution variable is initialized to `valueLeft`:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = valueLeft)
```

The `SteadyConvectionDiffusionScEquation` object is used to create the equation. It needs to be passed a convection term instantiator as follows:

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> from fipy.solvers.linearCGSSolver import LinearCGSSolver
>>> from fipy.equations.stdyConvDiffScEquation import SteadyConvectionDiffusionScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...      var = var,
...      diffusionCoeff = diffCoeff,
...      convectionCoeff = convCoeff,
...      solver = LinearCGSSolver(tolerance = 1.e-15, steps = 2000),
...      convectionScheme = ExponentialConvectionTerm,
...      boundaryConditions = boundaryConditions
...      )
```

More details of the benefits and drawbacks of each type of convection term can be found in the numerical section of the manual. Essentially the `ExponentialConvectionTerm` and `PowerLawConvectionTerm` will both handle most types of convection diffusion cases with the `PowerLawConvectionTerm` being more efficient.

We iterate to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> it.timestep()
```

and test the solution against the analytical result

$$\phi = \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:,axis]
>>> import Numeric
>>> CC = 1. - Numeric.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - Numeric.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> Numeric.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

6.2 Module examples.convection.exponential1DSource.input

Like `examples/diffusion/convection/exponential1D/input.py` this example solves a steady-state convection-diffusion equation, but adds a constant source, $S_0 = 1$, such that

$$\nabla \cdot (D\nabla\phi + \vec{u}\phi) + S_0 = 0.$$

Here, the axes are reversed

```
>>> nx = 1
>>> ny = 1000
```

and
$vecu = (0, 10)$ such that

```
>>> diffCoeff = 1.
>>> convCoeff = (0., 10.)
>>> sourceCoeff = 1.
```

We define a 1D mesh

```
>>> L = 10.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } y = 0, \\ 1 & \text{at } y = L, \end{cases}$$

or

```
>>> valueBottom = 0.
>>> valueTop = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesTop(), valueTop),
...     FixedValue(mesh.getFacesBottom(), valueBottom),
...     FixedFlux(mesh.getFacesRight(), 0.),
...     FixedFlux(mesh.getFacesLeft(), 0.)
...     )
```

The solution variable is initialized to `valueBottom`:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = valueBottom)
```

We define the convection-diffusion equation with source

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> from fipy.equations.stdyConvDiffScEquation import SteadyConvectionDiffusionScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...      var = var,
...      diffusionCoeff = diffCoeff,
...      convectionCoeff = convCoeff,
...      sourceCoeff = sourceCoeff,
...      solver = LinearLUSolver(tolerance = 1.e-15),
...      convectionScheme = ExponentialConvectionTerm,
...      boundaryConditions = boundaryConditions
...      )
```

iterate one implicit timestep to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
>>> it.timestep()
```

and test the solution against the analytical result:

$$\phi = -\frac{S_0 y}{u_y} + \left(1 + \frac{S_0 y}{u_y}\right) \frac{1 - \exp(-u_y y/D)}{1 - \exp(-u_y L/D)}$$

or

```
>>> axis = 1
>>> y = mesh.getCellCenters()[:,axis]
>>> AA = -sourceCoeff * y / convCoeff[axis]
>>> BB = 1. + sourceCoeff * L / convCoeff[axis]
>>> import Numeric
>>> CC = 1. - Numeric.exp(-convCoeff[axis] * y / diffCoeff)
>>> DD = 1. - Numeric.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = AA + BB * CC / DD
>>> Numeric.allclose(analyticalArray, var, rtol = 1e-4, atol = 1e-4)
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

Chapter 7

Phase Field Examples

Solidification Examples

The following examples exhibit various phenomena in solidification, including dendritic growth and grain impingement. Further discussion of the models and algorithms can be found in reference [13].

7.1 Module examples.phase.anisotropy.input

In this example we solve a coupled phase and temperature equation to model solidification, and eventually dendritic growth, from a circular seed in a 2D mesh:

```
>>> numberOfCells = 40
>>> Length = numberOfCells * 2.5 / 100.
>>> nx = numberOfCells
>>> ny = numberOfCells
>>> dx = Length / nx
>>> dy = Length / ny
>>> radius = Length / 4.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx,dy,nx,ny)
```

Dendritic growth will not be observed with this small test system. If you wish to see dendritic growth reset the following parameters: `numberOfCells = 200`, `steps = 10000`, `radius = Length / 80`.

The governing equation for the phase field is given by:

$$\tau_\phi \frac{\partial \phi}{\partial t} = \alpha^2 \nabla^2 \phi + \phi(1-\phi) m_2(\phi, T) - 2s\phi|\nabla\theta| - \epsilon^2 \phi |\nabla\theta|^2$$

where

$$m_2(\phi, T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 T)$$

and the governing equation for temperature is given by:

$$\frac{\partial T}{\partial t} = D_T \nabla^2 T + \frac{\partial \phi}{\partial t}$$

Here the phase and temperature equations are solved with an explicit and implicit technique, respectively.

The parameters for these equations are

```
>>> timeStepDuration = 5e-5
>>> phaseParameters = {
...     'tau'                   : 3e-4,
...     'epsilon'               : 0.008,
...     's'                     : 0.01,
...     'alpha'                 : 0.015,
...     'anisotropy'            : 0.02,
...     'symmetry'              : 4.,
...     'kappa 1'               : 0.9,
...     'kappa 2'               : 20.
...     }
>>> temperatureParameters = {
...     'timeStepDuration' : timeStepDuration,
...     'temperature diffusion' : 2.25,
...     'latent heat'           : 1.,
...     'heat capacity'         : 1.
...     }
```

The variable `theta` represents the orientation of the crystal. In this example, it is constant and thus does not affect the solution.

```
>>> from fipy.models.phase.theta.modularVariable import ModularVariable
>>> theta = ModularVariable(
...     name = 'Theta',
...     mesh = mesh
...     )
```

The `phase` variable is `0` for a liquid and `1` for a solid. Here we build an example `phase` variable, initialized as a liquid,

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(
...     name = 'PhaseField',
...     mesh = mesh,
...     value = 0.,
...     hasOld = 1)
```

The `hasOld` flag keeps the old value of the variable. This is necessary for a transient solution. In this example we wish to set up an interior region that is solid. A value of `1` is assigned to the `phase` variable on a patch defined by the method:

```
>>> def circleCells(cell,L = Length):
...     x = cell.getCenter()
...     r = radius
...     c = (Length / 2., Length / 2.)
...     if (x[0] - c[0])**2 + (x[1] - c[1])**2 < r**2:
...         return 1
...     else:
...         return 0
```

This method is passed to `mesh.getCells(filter = circleCells)` which filters out the required cells.

```
>>> interiorCells = mesh.getCells(filter = circleCells)
>>> phase.setValue(1.,interiorCells)
```

The temperature field is initialized to a value of `-0.4` throughout:

```
>>> temperature = CellVariable(
...     name = 'Theta',
...     mesh = mesh,
...     value = -0.4,
...     hasOld = 1
...     )
```

For both equations, zero flux boundary conditions apply to the exterior of the mesh

```
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryCondition = FixedFlux(mesh.getExteriorFaces(), 0.)
```

The `phase` equation requires a `mPhi` instantiator to represent $m_2(\phi, T)$ above

```
>>> from fipy.models.phase.phase.type2MPhiVariable import Type2MPhiVariable
```

The `phase` equation is solved with an iterative conjugate gradient solver

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
```

and requires access to the `theta` and `temperature` variables

```
>>> from fipy.models.phase.phase.phaseEquation import PhaseEquation
>>> phaseEq = PhaseEquation(
...     phase,
...     mPhi = Type2MPhiVariable,
...         solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 1000
...     ),
...     boundaryConditions=(boundaryCondition,),
...     parameters = phaseParameters,
...     fields = {
...         'theta' : theta,
...         'temperature' : temperature
...     }
...     )
```

The `temperature` equation is also solved with an iterative conjugate gradient solver and requires access to the `phase` variable

```
>>> from fipy.models.phase.temperature.temperatureEquation import TemperatureEquation
>>> temperatureEq = TemperatureEquation(
...     temperature,
...     solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 1000
...     ),
...     boundaryConditions=(boundaryCondition,),
...     parameters = temperatureParameters,
...     fields = {
...         'phase' : phase
...     }
...     )
```

If we are running this example interactively, we create viewers for the phase and temperature fields

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     phaseViewer = Grid2DGistViewer(var = phase)
...     temperatureViewer = Grid2DGistViewer(var = temperature,
...                                          minVal = -0.5, maxVal =0.5)
...     phaseViewer.plot()
...     temperatureViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((phaseEq, temperatureEq))
>>> steps = 10
>>> for i in range(steps):
...     it.timestep(dt = timeStepDuration)
...     if i%10 == 0 and __name__ == '__main__':
...         phaseViewer.plot()
...         temperatureViewer.plot()
```

The solution is compared with test data. The test data was created for `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `test.gz` extracts the data and compares it with the `phase` variable.

```
>>> import os
>>> testFile = 'test.gz'
>>> import examples.phase.anisotropy
>>> gzfile = 'gunzip --fast -c < %s/%s'
>>> gzfile = gzfile%(examples.phase.anisotropy.__path__[0], testFile)
>>> filestream=os.popen(gzfile,'r')
>>> import cPickle
>>> testData = cPickle.load(filestream)
>>> filestream.close()
>>> import Numeric
>>> phase =  Numeric.array(phase)
>>> testData = Numeric.reshape(testData, phase.shape)
>>> Numeric.allclose(phase, testData, rtol = 1e-10, atol = 1e-10)
1
```

7.2 Module examples.phase.impingement.mesh40x1.input

In this example we solve a coupled phase and orientation equation on a one dimensional grid

```
>>> nx = 40
>>> ny = 1
>>> Lx = 2.5 * nx / 100.
>>> Ly = 2.5 * ny / 100.
>>> dx = Lx / nx
>>> dy = Ly / ny
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx,dy,nx,ny)
```

This problem simulates the wet boundary that forms between grains of different orientations. The phase equation is given by

$$\tau_\phi \frac{\partial \phi}{\partial t} = \alpha^2 \nabla^2 \phi + \phi(1-\phi) m_1(\phi, T) - 2s\phi|\nabla\theta| - \epsilon^2 \phi |\nabla\theta|^2$$

where

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1-\phi)$$

and the orientation equation is given by

$$P(\epsilon|\nabla\theta|)\tau_\theta \phi^2 \frac{\partial \theta}{\partial t} = \nabla \cdot \left[\phi^2 \left(\frac{s}{|\nabla\theta|} + \epsilon^2\right) \nabla\theta\right]$$

where

$$P(w) = 1 - \exp(-\beta w) + \frac{\mu}{\epsilon}\exp(-\beta w)$$

The initial conditions for this problem are set such that $\phi = 1$ for $0 \le x \le L_x$ and

$$\theta = \begin{cases} 1 & \text{for } 0 \le x < L_x/2, \\ 0 & \text{for } L_x/2 \le x \le L_x. \end{cases}$$

Here the phase and orientation equations are solved with an explicit and implicit technique respectively.

The parameters for these equations are

```
>>> timeStepDuration = 0.02
>>> phaseParameters = {
...    'tau'                   : 0.1,
...     }
>>> thetaParameters = {
...     'small value'           : 1e-6,
...     'beta'                  : 1e5,
...     'mu'                    : 1e3,
...     'tau'                   : 0.01,
...     'gamma'                 : 1e3
...     }
```

with the shared parameters

```
>>> sharedPhaseThetaParameters = {
...     'epsilon'               : 0.008,
...     's'                     : 0.01,
...     'anisotropy'            : 0.0,
...     'alpha'                 : 0.015,
...     'symmetry'              : 4.
...     }
>>> for key in sharedPhaseThetaParameters.keys():
...     phaseParameters[key] = sharedPhaseThetaParameters[key]
...     thetaParameters[key] = sharedPhaseThetaParameters[key]
```

The system is held isothermal at

```
>>> temperature = 1.
```

and is initially solid everywhere

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(
...     name = 'PhaseField',
...     mesh = mesh,
...     value = 1.
...     )
```

The left and right halves of the domain are given different orientations

```
>>> from fipy.models.phase.theta.modularVariable import ModularVariable
>>> theta = ModularVariable(
...     name = 'Theta',
...     mesh = mesh,
...     value = 1.,
...     hasOld = 1
...     )
>>> theta.setValue(0., mesh.getCells(filter = lambda cell: cell.getCenter()[0] > Lx / 2.))
```

For both equations, zero flux boundary conditions apply to the exterior of the mesh

```
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryCondition = FixedFlux(mesh.getExteriorFaces(), 0.)
```

The `phase` equation requires a `mPhi` instantiator to represent $m_1(\phi, T)$ above

```
>>> from fipy.models.phase.phase.type1MPhiVariable import Type1MPhiVariable
```

The `phase` equation is solved with an iterative conjugate gradient solver

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
```

and requires access to the `theta` and `temperature` variables

```
>>> from fipy.models.phase.phase.phaseEquation import PhaseEquation
>>> phaseEq = PhaseEquation(
...     phase,
...     mPhi = Type1MPhiVariable,
...     solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 1000
...     ),
...     boundaryConditions=(boundaryCondition,),
...     parameters = phaseParameters,
...     fields = {
...         'theta' : theta,
...         'temperature' : temperature
...     }
...     )
```

The `theta` equation is also solved with an iterative conjugate gradient solver and requires access to the `phase` variable

```
>>> from fipy.models.phase.theta.thetaEquation import ThetaEquation
>>> thetaEq = ThetaEquation(
...     var = theta,
...     solver = LinearPCGSolver(
...             tolerance = 1.e-15,
...             steps = 2000
...     ),
...     boundaryConditions = (boundaryCondition,),
...     parameters = thetaParameters,
...     fields = {
...         'phase' : phase
...     }
...     )
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     phaseViewer = Grid2DGistViewer(var = phase, palette = 'rainbow.gp',
...                                    minVal = 0., maxVal = 1., grid = 0)
...     from Numeric import pi
...     thetaProd = -pi + phase * (theta + pi)
...     thetaProductViewer = Grid2DGistViewer(var = thetaProd , palette = 'rainbow.gp',
...                                           minVal = -pi, maxVal = pi, grid = 0)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((thetaEq, phaseEq))
>>> steps = 10
>>> for i in range(steps):
...     it.timestep(dt = timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared with test data. The test data was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `test.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> testFile = 'test.gz'
>>> import examples.phase.impingement.mesh40x1
>>> gzfile = 'gunzip --fast -c < %s/%s'
>>> gzfile = gzfile%(examples.phase.impingement.mesh40x1.__path__[0], testFile)
>>> filestream=os.popen(gzfile,'r')
>>> import cPickle
>>> testData = cPickle.load(filestream)
>>> filestream.close()
>>> import Numeric
>>> theta =  Numeric.array(theta)
>>> testData = Numeric.reshape(testData, theta.shape)
>>> Numeric.allclose(theta, testData, rtol = 1e-10, atol = 1e-10)
1
```

7.3 Module examples.phase.impingement.mesh20x20.base

In the following examples, we solve the same set of equations as in:

```
$ examples/phase/impingement/mesh40x1/input.py
```

with different initial conditions and a 2D mesh:

```
>>> nx = 20
>>> ny = 20
>>> Lx = 2.5 * nx / 100.
>>> Ly = 2.5 * ny / 100.
>>> dx = Lx / nx
>>> dy = Ly / ny
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx,dy,nx,ny)
```

The initial conditions are given by $\phi = 1$ and

$$\theta = \begin{cases} \frac{2\pi}{3} & \text{for } x^2 - y^2 < L/2, \\ \frac{-2\pi}{3} & \text{for } (x-L)^2 - y^2 < L/2, \\ \frac{-2\pi}{3} + 0.3 & \text{for } x^2 - (y-L)^2 < L/2, \\ \frac{2\pi}{3} & \text{for } (x-L)^2 - (y-L)^2 < L/2. \end{cases}$$

This defines four solid regions with different orientations. Solidification occurs and then boundary wetting occurs where the orientation varies.

The parameters for this example are

```
>>> timeStepDuration = 0.02
>>> phaseParameters = {
...     'tau'                   : 0.1,
...     'time step duration'    : timeStepDuration
...     }
>>> thetaParameters = {
...     'small value'            : 1e-6,
...     'beta'                   : 1e5,
...     'mu'                     : 1e3,
...     'tau'                    : 0.01,
...     'gamma'                  : 1e3
...     }
```

with the shared parameters

```
>>> sharedPhaseThetaParameters = {
...     'epsilon'                : 0.008,
...     's'                      : 0.01,
...     'anisotropy'             : 0.0,
...     'alpha'                 : 0.015,
...     'symmetry'              : 4.
...     }
>>> for key in sharedPhaseThetaParameters.keys():
...     phaseParameters[key] = sharedPhaseThetaParameters[key]
...     thetaParameters[key] = sharedPhaseThetaParameters[key]
```

This time, the system is held isothermal at

```
>>> temperature = 10.
```

and is initialized to liquid everywhere

```
>>> from fipy.variables.cellVariable import CellVariable
>>> phase = CellVariable(
...     name = 'PhaseField',
...     mesh = mesh,
...     value = 0.
...     )
```

The orientation is initialized to a uniform value to denote the randomly oriented liquid phase

```
>>> from Numeric import pi
>>> from fipy.models.phase.theta.modularVariable import ModularVariable
>>> theta = ModularVariable(
...     name = 'Theta',
...     mesh = mesh,
...     value = -pi + 0.0001,
...     hasOld = 1
...     )
```

Four different solid circular domains are created at each corner of the domain with appropriate orientations

```
>>> def cornerCircle(cell):
...     x = cell.getCenter()[0]
...     y = cell.getCenter()[1]
...     if ((x - a)**2 + (y - b)**2) < (Lx / 2.)**2:
...         return 1
...     else:
...         return 0

>>> for a, b, thetaValue in ((0., 0.,  2. * pi / 3.),
...                          (Lx, 0., -2. * pi / 3.),
...                          (0., Ly, -2. * pi / 3. + 0.3),
...                          (Lx, Ly,  2. * pi / 3.)):
...     cells = mesh.getCells(filter = cornerCircle)
...     phase.setValue(1., cells)
...     theta.setValue(thetaValue, cells)
```

For both equations, zero flux boundary conditions apply to the exterior of the mesh

```
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryCondition = FixedFlux(mesh.getExteriorFaces(), 0.)
```

The `phase` equation requires a `mPhi` instantiator to represent $m_1(\phi, T)$ above

```
>>> from fipy.models.phase.phase.type1MPhiVariable import Type1MPhiVariable
```

The `phase` equation is solved with an iterative conjugate gradient solver

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
```

and requires access to the `theta` and `temperature` variables

```
>>> from fipy.models.phase.phase.phaseEquation import PhaseEquation
>>> phaseEq = PhaseEquation(
...     phase,
...     mPhi = Type1MPhiVariable,
...     solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 1000
...     ),
...     boundaryConditions=(boundaryCondition,),
...     parameters = phaseParameters,
...     fields = {
...         'theta' : theta,
...         'temperature' : temperature
...     }
...     )
```

The `theta` equation is also solved with an iterative conjugate gradient solver and requires access to the `phase` variable

```
>>> from fipy.models.phase.theta.thetaEquation import ThetaEquation
>>> thetaEq = ThetaEquation(
...     var = theta,
...     solver = LinearPCGSolver(
...             tolerance = 1.e-15,
...             steps = 2000
...     ),
...     boundaryConditions = (boundaryCondition,),
...     parameters = thetaParameters,
...     fields = {
...         'phase' : phase
...     }
...     )
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
```

```
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     phaseViewer = Grid2DGistViewer(var = phase, palette = 'rainbow.gp',
...                                    minVal = 0., maxVal = 1., grid = 0)
...     from Numeric import pi
...     thetaProd = -pi + phase * (theta + pi)
...     thetaProductViewer = Grid2DGistViewer(var = thetaProd , palette = 'rainbow.gp',
...                                           minVal = -pi, maxVal = pi, grid = 0)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

The solution will be tested against data that was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `test.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> testFile = 'test.gz'
>>> import examples.phase.impingement.mesh20x20
>>> gzfile = 'gunzip --fast -c < %s/%s'
>>> gzfile = gzfile%(examples.phase.impingement.mesh20x20.__path__[0], testFile)
>>> filestream=os.popen(gzfile,'r')
>>> import cPickle
>>> testData = cPickle.load(filestream)
>>> filestream.close()
>>> import Numeric
>>> testData = Numeric.reshape(testData, Numeric.array(theta).shape)
```

Finally, we create an iterator

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((thetaEq, phaseEq))
>>> steps = 10
```

The preceding initialization steps are used in the next few examples.

7.3.1 Functions

script()

Return the documentation for this module as a script that can be invoked to initialize other scripts.

7.4 Module examples.phase.impingement.mesh20x20.input

We initialize the system by running the base script

```
>>> import examples.phase.impingement.mesh20x20.base
>>> exec(examples.phase.impingement.mesh20x20.base.script())
```

We iterate the solution in time, plotting as we go if running interactively,

```
>>> for i in range(steps):
...     it.timestep(dt = timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared against Ryo Kobayashi's test data

```
>>> theta.allclose(testData, rtol = 1e-10, atol = 1e-10)
1
```

7.5 Module examples.phase.impingement.restart.input

In this example we solve the same set of equations as in `examples/phase/impingement/mesh20x20/input.py` but a restart method is demonstrated.

We again initialize the system by running the base script

```
>>> import examples.phase.impingement.mesh20x20.base
>>> exec(examples.phase.impingement.mesh20x20.base.script())
```

but this time we only iterate for half as many time steps

```
>>> for i in range(steps/2):
...     it.timestep(dt = timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

We confirm that the solution has not yet converged to that given by Ryo Kobayashi's FORTRAN code:

```
>>> theta.allclose(testData, rtol = 1e-10, atol = 1e-10)
0
```

We save the variables to disk

```
>>> import fipy.tools.dump as dump
>>> import tempfile
>>> import os
>>> tmp = tempfile.gettempdir()
>>> fileName = os.path.join(tmp, 'data')
>>> dump.write({'phase' : phase, 'theta' : theta, 'mesh' : mesh}, fileName)
```

and then recall them to test the data pickling mechanism

```
>>> data = dump.read(fileName)
>>> newPhase = data['phase']
>>> newTheta = data['theta']
>>> newMesh = data['mesh']
```

We rebuild the equations:

```
>>> newThetaEq = ThetaEquation(
...     var = newTheta,
...     solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 2000
...     ),
...     boundaryConditions = (
...         FixedFlux(newMesh.getExteriorFaces(), 0.),
...     ),
...     parameters = thetaParameters,
...     fields = {
...         'phase' : newPhase
...     }
... )
```

```
>>> newPhaseEq = PhaseEquation(
...     var = newPhase,
...     mPhi = Type1MPhiVariable,
...     solver = LinearPCGSolver(
...         tolerance = 1.e-15,
...         steps = 1000
...     ),
...     boundaryConditions = (
...         FixedFlux(newMesh.getExteriorFaces(), 0.),
...      ),
...      parameters = phaseParameters,
...      fields = {
...          'theta' : newTheta,
...          'temperature' : temperature
...      }
... )
```

the iterator:

```
>>> newIt = Iterator((newThetaEq, newPhaseEq))
```

and, if the example is run interactively, we recreate the viewers:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     newPhaseViewer = Grid2DGistViewer(var = newPhase, palette = 'rainbow.gp',
...                                       minVal = 0., maxVal = 1., grid = 0)
...     from Numeric import pi
...     newThetaProd = -pi + phase * (newTheta + pi)
...     newThetaProductViewer = \
...         Grid2DGistViewer(var = newThetaProd , palette = 'rainbow.gp',
...                          minVal = -pi, maxVal = pi, grid = 0)
...     newPhaseViewer.plot()
...     newThetaProductViewer.plot()
```

and finish doing the iterations

```
>>> for i in range(steps - steps/2):
...     newIt.timestep(dt = timeStepDuration)
...     if __name__ == '__main__':
...         newPhaseViewer.plot()
...         newThetaProductViewer.plot()
```

Finally, we check that the results have at last converged to Ryo Kobayashi's FORTRAN code:

```
>>> newTheta.allclose(testData, rtol = 1e-10, atol = 1e-10)
1
```

Electrochemistry Examples

The following examples exhibit various parts of a model to study electrochemical interfaces. In a pair of papers, Guyer, Boettinger, Warren and McFadden [14, 15] have shown that an electrochemical interface can be modeled by an equation for the phase field ξ

$$\frac{\partial \xi}{\partial t} = M_\xi \kappa_\xi \nabla^2 \xi - M_\xi \sum_{j=1}^{n} C_j \left[p'(\xi) \Delta\mu_j^\circ + g'(\xi) W_j\right] + M_\xi \frac{\epsilon'(\xi)}{2} \left(\nabla \phi\right)^2$$

a set of diffusion equations for the concentrations C_j, for $j = 2, \ldots, n-1$, of the substitutional elements

$$\begin{aligned}\frac{\partial C_j}{\partial t} &= D_j \nabla^2 C_j \\ &\qquad + D_j \nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=2 \\ k \neq j}}^{n-1} C_k} \left\{ \sum_{\substack{i=2 \\ i \neq j}}^{n-1} \nabla C_i + C_n \left[p'(\xi) \Delta\mu_{jn}^\circ + g'(\xi) W_{jn}\right] \nabla \xi + C_n z_{jn} \nabla \phi \right\}\end{aligned}$$

a diffusion equation for the concentration C_{e^-} of electrons

$$\frac{\partial C_{\mathrm{e}^-}}{\partial t} = D_{\mathrm{e}^-} \nabla^2 C_{\mathrm{e}^-} + D_{\mathrm{e}^-} \nabla \cdot C_{\mathrm{e}^-} \left\{ \left[p'(\xi) \Delta\mu_{\mathrm{e}^-}^\circ + g'(\xi) W_{\mathrm{e}^-}\right] \nabla \xi + z_{\mathrm{e}^-} \nabla \phi \right\}$$

and Poisson's equation for the electrostatic potential ϕ

$$\nabla \cdot (\epsilon \nabla \phi) + \rho = 0$$

M_ξ is the phase field mobility, κ_ξ is the phase field gradient energy coefficient, $p'(\xi) = 30\xi^2 \left(1 - \xi\right)^2$, and $g'(\xi) = 2\xi \left(1 - \xi\right) \left(1 - 2\xi\right)$. For a given species j, $\Delta\mu_j^\circ$ is the standard chemical potential difference between the electrode and electrolyte for a pure material, W_j is the magnitude of the energy barrier in the double-well free energy function, z_j is the valence, and D_j is the self diffusivity. $\Delta\mu_{jn}^\circ$, W_{jn}, and z_{jn} are the differences of the respective quantities $\Delta\mu_j^\circ$, W_j, and z_j between substitutional species j and the solvent species n. The total charge is denoted by $\rho \equiv \sum_{j=1}^{n} z_j C_j$.

The module `fipy.models.elphf` has been developed to solve this coupled set of equations. Although unresolved stiffnesses make the full solution intractable in FiPy, we can demonstrate the use of various parts of the `elphf` module.

7.6 Module examples.elphf.phase.input1D

A simple 1D phase-field problem to test the `PhaseEquation` element of ElPhF.

The single-component phase field governing equation can be represented as

$$\frac{1}{M_\xi}\frac{\partial \xi}{\partial t} = \kappa_\xi \nabla^2 \xi - 2\xi(1-\xi)(1-2\xi)W$$

where ξ is the phase field, t is time, M_ξ is the phase field mobility, κ_ξ is the phase field gradient energy coefficient, and W is the phase field barrier energy. We solve the problem on a 1D mesh

```
>>> nx = 400
>>> dx = 0.01
>>> ny = 1
>>> dy = dx
>>> L = nx * dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

Rather than rewriting the same code in every electrochemistry example, we use the ElPhF module

```
>>> import fipy.models.elphf.elphf as elphf
```

to build the approriate variable fields from

```
>>> parameters = {
...     'time step duration': 10000,
...     'phase': {
...             'name': "xi",
...             'mobility': 1.,
...             'gradient energy': 0.025,
...             'value': 1.
...     },
...     'solvent': {
...             'standard potential': 0.,
...             'barrier height': 1.
...     }
... }

>>> fields = elphf.makeFields(mesh = mesh, parameters = parameters)
```

We separate the phase field into electrode and electrolyte regimes

```
>>> setCells = mesh.getCells(filter = lambda cell: cell.getCenter()[0] > L/2)
>>> fields['phase'].setValue(1.)
>>> fields['phase'].setValue(0.,setCells)
```

We use the ElPhF module again to create governing equations from the fields

```
>>> equations = elphf.makeEquations(mesh = mesh,
...                                 fields = fields,
...                                 parameters = parameters)
```

If we are running interactively, we will want to see the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = (fields['phase'],))
...     viewer.plot()
```

Now, we iterate to equilibrium, plotting as we go

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator(equations = equations)
>>> for i in range(50):
...     it.timestep(1)
...     if __name__ == '__main__':
...         viewer.plot()
```

The phase field has the expected analytical form

$$\xi(x) = \frac{1}{2}(1 - \tanh\frac{x - L/2}{2d})$$

where the interfacial thickness is given by $d = \sqrt{\kappa_\xi / W}$. We verify that the correct equilibrium solution is attained

```
>>> x = mesh.getCellCenters()[:,0]

>>> import Numeric
>>> d = Numeric.sqrt(parameters['phase']['gradient energy']
...     / (parameters['solvent']['barrier height']))
>>> analyticalArray = (1. - Numeric.tanh((x - L/2.)/(2.*d))) / 2.

>>> fields['phase'].allclose(analyticalArray, rtol = 1e-4, atol = 1e-4)
1
```

7.7 Module examples.elphf.diffusion.input1D

A simple 1D three-component diffusion problem to test the `ConcentrationEquation` element of ElPhF. The diffusion equation for each species in single-phase multicomponent system can be expressed as

$$\frac{\partial C_j}{\partial t} = D_{jj}\nabla^2 C_j + D_{jj}\nabla \cdot \left[\frac{C_j}{1 - \sum_{\substack{k=2 \\ k \neq j}}^{n-1} C_k} \sum_{\substack{i=2 \\ i \neq j}}^{n-1} \nabla C_i \right]$$

where C_j is the concentration of the j^{th} species, t is time, D_{jj} is the self-diffusion coefficient of the j^{th} species, and $\sum_{\substack{i=2 \\ i \neq j}}^{n-1}$ represents the summation over all substitutional species in the system, excluding the solvent and the component of interest.

We solve the problem on a 1D mesh

```
>>> nx = 40
>>> dx = 1.
>>> ny = 1
>>> dy = 1
>>> L = nx * dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

The parameters for this problem are

```
>>> parameters = {
...     'time step duration': 10000,
...     'solvent': {
...         'standard potential': 0.,
...         'barrier height': 0.
...     }
... }
```

The ElPhF module allows the modeling of an arbitrary number of components, specified simply by providing a `Tuple` of species parameters

```
>>> parameters['substitutionals'] = (
...     {
...         'name': "c1",
...         'diffusivity': 1.,
...         'standard potential': 1.,
...         'barrier height': 1.
...     },
...     {
...         'name': "c2",
...         'diffusivity': 1.,
...         'standard potential': 1.,
...         'barrier height': 1.
...     }
... )
```

We use ElPhF to create the variable fields

```
>>> import fipy.models.elphf.elphf as elphf
>>> fields = elphf.makeFields(mesh = mesh, parameters = parameters)
```

and we separate the solution domain into two different concentration regimes

```
>>> setCells = mesh.getCells(filter = lambda cell: cell.getCenter()[0] > L/2)
>>> fields['substitutionals'][0].setValue(0.3)
>>> fields['substitutionals'][0].setValue(0.6,setCells)
>>> fields['substitutionals'][1].setValue(0.6)
>>> fields['substitutionals'][1].setValue(0.3,setCells)
```

We use ElPhF again to create the governing equations for the fields

```
>>> equations = elphf.makeEquations(mesh = mesh,
...                                 fields = fields,
...                                 parameters = parameters
... )
```

If we are running interactively, we create a viewer to see the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = (fields['solvent'],) + fields['substitutionals'],
...                           limits = ('e', 'e', 0, 1))
...     viewer.plot()
```

Note

the `Gist1DViewer` is capable of plotting multiple fields

Now, we iterate the problem to equilibrium, plotting as we go

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator(equations = equations)
>>> for i in range(40):
...     it.timestep(dt = parameters['time step duration'])
...     if __name__ == '__main__':
...         viewer.plot()
```

Since there is nothing to maintain the concentration separation in this problem, we verify that the concentrations have become uniform

```
>>> fields['substitutionals'][0].allclose(0.45, rtol = 1e-7, atol = 1e-7)
1
>>> fields['substitutionals'][1].allclose(0.45, rtol = 1e-7, atol = 1e-7)
1
```

7.8 Module examples.elphf.diffusion.input1Ddimensional

In this example, we present the same three-component diffusion problem introduced in `examples/elphf/input1D.py` but we demonstrate FiPy's facility to use dimensional quantities.

```
>>> from fipy.tools.dimensions.physicalField import PhysicalField
```

We solve the problem on a 40 mm long 1D mesh

```
>>> nx = 40
>>> dx = PhysicalField(1.,"mm")
>>> ny = 1
>>> dy = 1
>>> L = nx * dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

The dimensional parameters for this problem are

```
>>> parameters = {
...     'time step duration': "1000 s",
...     'solvent': {
...         'standard potential': 0.,
...         'barrier height': 0.
...     }
... }
>>> parameters['substitutionals'] = (
...     {
...         'name': "c1",
...         'diffusivity': "1.e-9 m**2/s",
...         'standard potential': 1.,
...         'barrier height': 1.
...     },
...     {
...         'name': "c2",
...         'diffusivity': "1.e-9 m**2/s",
...         'standard potential': 1.,
...         'barrier height': 1.
...     }
... )
```

We use ElPhF to create the variable fields

```
>>> import fipy.models.elphf.elphf as elphf
>>> fields = elphf.makeFields(mesh = mesh, parameters = parameters)
```

and we separate the solution domain into two different concentration regimes

```
>>> setCells = mesh.getCells(filter = lambda cell:
...                          cell.getCenter()[0] > mesh.getPhysicalShape()[0]/2)
>>> fields['substitutionals'][0].setValue("0.3 mol/m**3")
>>> fields['substitutionals'][0].setValue("0.6 mol/m**3",setCells)
>>> fields['substitutionals'][1].setValue("0.6 mol/m**3")
>>> fields['substitutionals'][1].setValue("0.3 mol/m**3",setCells)
```

We use ElPhF again to create the governing equations for the fields

```
>>> equations = elphf.makeEquations(mesh = mesh,
...                                 fields = fields,
...                                 parameters = parameters
... )
```

If we are running interactively, we create a viewer to see the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = (fields['solvent'],) + fields['substitutionals'],
...                             limits = ('e', 'e', 0, 1))
...     viewer.plot()
```

Now, we iterate the problem to equilibrium, plotting as we go

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator(equations = equations)
>>> for i in range(40):
...     it.timestep(dt = 10000)
...     if __name__ == '__main__':
...         viewer.plot()
```

Since there is nothing to maintain the concentration separation in this problem, we verify that the concentrations have become uniform

```
>>> fields['substitutionals'][0].getScaled().allclose("0.45 mol/m**3",
...     atol = "1e-7 mol/m**3", rtol = 1e-7)
1
>>> fields['substitutionals'][1].getScaled().allclose("0.45 mol/m**3",
...     atol = "1e-7 mol/m**3", rtol = 1e-7)
1
```

Note

The absolute tolerance `atol` must be in units compatible with the value to be checked, but the relative tolerance `rtol` is dimensionless.

7.9 Module examples.elphf.poisson.input1DrightCharge

A simple problem to test the `PoissonEquation` element of ElPhF on a 1D mesh

```
>>> nx = 200
>>> dx = 0.01
>>> L = nx * dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dx, nx = nx, ny = 1)
```

The dimensionless Poisson equation is

$$\nabla \cdot (\epsilon \nabla \psi) = -\rho = -\sum_{j=1}^{n} z_j C_j$$

where ψ is the electrostatic potential, ϵ is the permittivity, ρ is the charge density, C_j is the concentration of the j^{th} component, and z_j is the valence of the j^{th} component.

We examine a fixed distribution of electrons with $z_{\text{e}^-} = -1$ and and we let the permittivity $\epsilon = 1$. In the ElPhF construction, electrons are treated as interstitial elements, which can diffuse freely without displacing other components

```
>>> parameters = {
...     'potential': {
...         'name': "psi",
...         'permittivity': 1.,
...     },
...     'interstitials': (
...         {
...             'name': "e-",
...             'valence': -1,
...             'diffusivity': 0
...         },
...     )
... }
```

We have set the diffusivity of electrons to zero to keep them from moving due to electromigration.

We again let the ElPhF module construct the appropriate fields and governing equations

```
>>> import fipy.models.elphf.elphf as elphf
>>> fields = elphf.makeFields(mesh = mesh, parameters = parameters)
>>> equations = elphf.makeEquations(mesh = mesh,
...                                 fields = fields,
...                                 parameters = parameters)
```

We segregate all of the electrons to one side of the domain

$$C_{\mathrm{e}^-} = \begin{cases} 0 & \text{for } x \leq L/2, \\ 1 & \text{for } x > L/2. \end{cases}$$

```
>>> setCells = mesh.getCells(filter = lambda cell: cell.getCenter()[0] > L/2.)
>>> fields['interstitials'][0].setValue(0.)
>>> fields['interstitials'][0].setValue(1.,setCells)
```

and iterate one implicit timestep to equilibrate the electrostatic potential

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator(equations = equations)
>>> it.timestep()
```

This problem has the analytical solution

$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

We verify that the correct equilibrium is attained

```
>>> x = mesh.getCellCenters()[:,0]

>>> import Numeric
>>> analyticalArray = Numeric.where(x < 1, -x, ((x-1)**2)/2 - x)
>>> fields['potential'].allclose(analyticalArray, rtol = 2e-5, atol = 2e-5)
1
```

If we are running the example interactively, we view the result

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = (fields['charge'], fields['potential']))
...     viewer.plot()
```

7.10 Module examples.elphf.phaseDiffusion.input1Dbinary

This example combines a phase field problem, as given in `examples/elphf/input1Dphase.py`, with a binary diffusion problem, such as described in the ternary example `examples/elphf/input1D.py`, on a 1D mesh

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dx, nx = nx, ny = 1)
```

The problem parameters are

```
>>> parameters = {
...     'time step duration': 10000,
...     'substitutional molar volume': 1,
...     'phase': {
...         'name': "xi",
...         'mobility': 1.,
...         'gradient energy': 0.1,
...         'value': 1
...     },
... }
```

The thermodynamic parameters are chosen to give a solid phase rich in the solute and a liquid phase rich in the solvent

```
>>> import Numeric
>>> parameters['solvent'] = {
...     'standard potential': Numeric.log(.7/.3),
...     'barrier height': 1.
... }
```

```
>>> parameters['substitutionals'] = (
...     {
...         'name': "c1",
...         'diffusivity': 1.,
...         'standard potential': Numeric.log(.3/.7),
...         'barrier height': parameters['solvent']['barrier height'],
...     },
... )
```

We again let the ElPhF module create the appropriate fields and equations

```
>>> import fipy.models.elphf.elphf as elphf
>>> fields = elphf.makeFields(mesh = mesh, parameters = parameters)
>>> equations = elphf.makeEquations(mesh = mesh,
...                                 fields = fields,
...                                 parameters = parameters)
```

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \le L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

or

```
>>> setCells = mesh.getCells(filter = lambda cell: cell.getCenter()[0] > L/2)
>>> fields['phase'].setValue(1.)
>>> fields['phase'].setValue(0.,setCells)
```

and with a uniform concentration field $C_1 = 0.5$. or

```
>>> fields['substitutionals'][0].setValue(0.5)
```

If running interactively, we create viewers to display the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gist1DViewer import Gist1DViewer
...
...     phaseViewer = Gist1DViewer(vars = (fields['phase'],))
...     concViewer = Gist1DViewer(vars = (fields['solvent'],) + fields['substitutionals'],
...                               limits = ('e', 'e', 0, 1))
...     phaseViewer.plot()
...     concViewer.plot()
```

This problem does not have an analytical solution, so after iterating to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator(equations = equations)
>>> for i in range(50):
...     it.timestep()
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         concViewer.plot()
```

we confirm that the far-field phases have remained separated

```
>>> ends = Numeric.take(fields['phase'], (0,-1))
>>> Numeric.allclose(ends, (1.0, 0.0), rtol = 2e-3, atol = 2e-3)
1
```

and that the concentration field has appropriately segregated into solute rich and solute poor phases.

```
>>> ends = Numeric.take(fields['substitutionals'][0], (0,-1))
>>> Numeric.allclose(ends, (0.7, 0.3), rtol = 2e-3, atol = 2e-3)
1
```

7.11 Module examples.elphf.phaseDiffusion.input1DternaryAndElectr

This example adds two more components to `examples/elphf/input1DphaseBinary.py` one of which is another substitutional species and the other represents electrons and diffuses interterstitially.

We start by defining a 1D mesh

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dx, nx = nx, ny = 1)
```

The problem parameters are

```
>>> parameters = {
...     'time step duration': 10000,
...     'substitutional molar volume': 1.,
...     'phase': {
...         'name': "xi",
...         'mobility': 1.,
...         'gradient energy': 0.025,
...         'value': 1.
...     }
... }
```

The thermodynamic parameters are chosen to give a solid phase rich in electrons and the solvent and a liquid phase rich in the two substitutional species

```
>>> import Numeric
>>> parameters['solvent'] = {
...     'standard potential': Numeric.log(.4/.6) - Numeric.log(1.3/1.4),
...     'barrier height': 1.
... }

>>> parameters['interstitials'] = (
...     {
...         'name': "c1",
...         'diffusivity': 1.,
...         'standard potential': Numeric.log(.3/.4) - Numeric.log(1.3/1.4),
...         'barrier height': 0.,
...     },
... )

>>> parameters['substitutionals'] = (
...     {
...         'name': "c2",
...         'diffusivity': 1.,
...         'standard potential': Numeric.log(.4/.3) - Numeric.log(1.3/1.4),
...         'barrier height': parameters['solvent']['barrier height'],
...     },
...     {
...         'name': "c3",
...         'diffusivity': 1.,
...         'standard potential': Numeric.log(.2/.1) - Numeric.log(1.3/1.4),
...         'barrier height': parameters['solvent']['barrier height'],
...     },
... )
```

We again let the ElPhF module create the appropriate fields and equations

```
>>> import fipy.models.elphf.elphf as elphf
>>> fields = elphf.makeFields(mesh = mesh, parameters = parameters)
>>> equations = elphf.makeEquations(mesh = mesh,
...                                 fields = fields,
...                                 parameters = parameters)
```

Once again, we start with a sharp phase boundary

```
>>> setCells = mesh.getCells(filter = lambda cell: cell.getCenter()[0] > L/2)
>>> fields['phase'].setValue(1.)
>>> fields['phase'].setValue(0.,setCells)
```

and with uniform concentration fields, with the interstitial concentration $C_1 = 0.35$ and the substitutional concentrations $C_2 = 0.35$ and $C_3 = 0.15$.

```
>>> fields['interstitials'][0].setValue(0.35)
>>> fields['substitutionals'][0].setValue(0.35)
>>> fields['substitutionals'][1].setValue(0.15)
```

If running interactively, we create viewers to display the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gist1DViewer import Gist1DViewer
...
...     phaseViewer = Gist1DViewer(vars = (fields['phase'],))
...     concViewer = Gist1DViewer(vars = (fields['solvent'],)
...                                    + fields['substitutionals']
...                                    + fields['interstitials'],
...                                limits = ('e', 'e', 0, 1))
...     phaseViewer.plot()
...     concViewer.plot()
```

Again, this problem does not have an analytical solution, so after iterating to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator(equations = equations)
>>> for i in range(50):
...     it.timestep()
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         concViewer.plot()
```

we confirm that the far-field phases have remained separated

```
>>> ends = Numeric.take(fields['phase'], (0,-1))
>>> Numeric.allclose(ends, (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the concentration fields has appropriately segregated into into their respective phases

```
>>> ends = Numeric.take(fields['interstitials'][0], (0,-1))
>>> Numeric.allclose(ends, (0.4, 0.3), rtol = 3e-3, atol = 3e-3)
1
>>> ends = Numeric.take(fields['substitutionals'][0], (0,-1))
>>> Numeric.allclose(ends, (0.3, 0.4), rtol = 3e-3, atol = 3e-3)
1
>>> ends = Numeric.take(fields['substitutionals'][1], (0,-1))
>>> Numeric.allclose(ends, (0.1, 0.2), rtol = 3e-3, atol = 3e-3)
1
```

Chapter 8

Level Set Examples

The Level Set Method (LSM) is a popular interface tracking method. Further details of the LSM and descriptions of the algorithms used in FiPy can be found in Sethian's Level Set book [16].

8.1 Module examples.levelSet.distanceFunction.oneD.input

Here we solve the level set equation in one dimension. The level set equation solves a variable so that its value at any point in the domain is the distance from the zero level set. This can be represented succinctly in the following equation with a boundary condition at the zero level set such that,

$$\frac{\partial \phi}{\partial x} = 1$$

with the boundary condition, $\phi = 0$ at $x = L/2$. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> dx = 0.5
>>> dy = 2.
>>> nx = 10
>>> ny = 1
>>> L = nx * dx
```

Construct the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

Construct a `distanceVariable` object. This object is required by the `distanceEquation`.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(name = 'level set variable',
...                        mesh = mesh,
...                        value = -1.)
```

The domain must be divided into positive and negative regions.

```
>>> positiveCells = mesh.getCells(filter = lambda cell: cell.getCenter()[0] < L / 2.)
>>> var.setValue(1.,positiveCells)
```

The `distanceEquation` is then constructed.

```
>>> from fipy.models.levelSet.distanceFunction.distanceEquation import DistanceEquation
>>> eqn = DistanceEquation(var)
```

The problem can then be solved by executing the `solve()` method of the equation.

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var = var, palette = 'rainbow.gp',
...                               minVal = -5., maxVal = 5.)
...     viewer.plot()
...     eqn.solve()
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> eqn.solve()
>>> import Numeric
>>> Numeric.allclose(var,
...     Numeric.array((9. * dx / 2., 7. * dx / 2., 5. * dx / 2.,
...     3. * dx / 2., dx / 2., -dx / 2., -3. * dx / 2., -5. * dx / 2.,
...     -7. * dx / 2., -9. * dx / 2.)))
1
```

8.2 Module examples.levelSet.distanceFunction.circle.input

Here we solve the level set equation in two dimensions for a circle. The 2D level set equation can be written,

$$|\nabla\phi| = 1$$

and the boundary condition for a circle is given by, $\phi = 0$ at $(x - L/2)^2 + (y - L/2)^2 = (L/4)^2$. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = nx * dx
>>> Ly = ny * dy
```

Construct the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

Construct a `distanceVariable` object. This object is required by the `distanceEquation`.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(name = 'level set variable',
...                        mesh = mesh,
...                        value = -1.)
```

The domain must be divided into positive and negative regions.

```
>>> positiveCells = mesh.getCells(filter = lambda cell:
...                   (cell.getCenter()[0] - Lx / 2.)**2 +
...                   (cell.getCenter()[1] - Ly / 2.)**2 <
...                   (Lx / 4.)**2)
>>> var.setValue(1.,positiveCells)
```

The `distanceEquation` is then constructed.

```
>>> from fipy.models.levelSet.distanceFunction.distanceEquation import DistanceEquation
>>> eqn = DistanceEquation(var)
```

The problem can then be solved by executing the `solve()` method of the equation.

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var = var, palette = 'rainbow.gp', minVal = -5., maxVal = 5.)
...     viewer.plot()
...     eqn.solve()
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> eqn.solve()
>>> dY = dy / 2.
>>> dX = dx / 2.
>>> mm = min (dX, dY)
>>> import Numeric
>>> m1 = dY * dX / Numeric.sqrt(dY**2 + dX**2)
>>> def evalCell(phix, phiy, dx, dy):
...     aa = dy**2 + dx**2
...     bb = -2 * ( phix * dy**2 + phiy * dx**2)
...     cc = dy**2 * phix**2 + dx**2 * phiy**2 - dx**2 * dy**2
...     sqr = Numeric.sqrt(bb**2 - 4. * aa * cc)
...     return ((-bb - sqr) / 2. / aa,  (-bb + sqr) / 2. / aa)
>>> v1 = evalCell(-dY, -m1, dx, dy)[0]
>>> v2 = evalCell(-m1, -dX, dx, dy)[0]
>>> v3 = evalCell(m1,  m1,  dx, dy)[1]
>>> v4 = evalCell(v3, dY, dx, dy)[1]
>>> v5 = evalCell(dX, v3, dx, dy)[1]
>>> import MA
>>> trialValues = MA.masked_values((
...     -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000,
...     -1000, -1000, -1000, -1000, -3*dY, -3*dY, -3*dY, -1000, -1000, -1000, -1000,
...     -1000, -1000, -1000, v1   , -dY  , -dY  , -dY  , v1   , -1000, -1000, -1000,
...     -1000, -1000, v2   , -m1  , m1   , dY   , m1   , -m1  , v2   , -1000, -1000,
...     -1000, -dX*3, -dX  , m1   ,  v3  , v4   , v3   , m1   , -dX  , -dX*3, -1000,
...     -1000, -dX*3, -dX  , dX   , v5   , -1000, v5   , dX   , -dX  , -dX*3, -1000,
...     -1000, -dX*3, -dX  , m1   , v3   , v4   , v3   , m1   , -dX  , -dX*3, -1000,
...     -1000, -1000, v2   , -m1  , m1   , dY   , m1   , -m1  , v2   , -1000, -1000,
...     -1000, -1000, -1000, v1   , -dY  , -dY  , -dY  , v1   , -1000, -1000, -1000,
...     -1000, -1000, -1000, -1000, -3*dY, -3*dY, -3*dY, -1000, -1000, -1000, -1000,
...     -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000, -1000),
...     -1000)
>>> MA.allclose(var, trialValues)
1
```

8.3 Module examples.levelSet.advection.mesh1D.input

This example first solves the distance function equation in one dimension:

$$|\nabla\phi| = 1$$

with $\phi = 0$ at $x = L/5$. The variable is then advected with,

$$\frac{\partial\phi}{\partial t} + \vec{u}\cdot\nabla\phi = 0$$

The scheme used in the `AdvectionTerm` preserves the `var` as a distance function.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> velocity = 1.
>>> dx = 1.
>>> dy = 1.
>>> nx = 10
>>> ny = 1
>>> timeStepDuration = 1.
>>> steps = 2
>>> L = nx * dx
>>> interfacePosition = L / 5.
```

Construct the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
```

Construct a `distanceVariable` object. This object is required by the `distanceEquation`.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(name = 'level set variable',
...                          mesh = mesh,
...                          value = -1.)
```

The domain must be divided into positive and negative regions.

```
>>> var.setValue(1.,  mesh.getCells(filter = lambda cell:
...                                 cell.getCenter()[0] > interfacePosition))
```

The `distanceEquation` is then constructed.

```
>>> from fipy.models.levelSet.distanceFunction.distanceEquation import DistanceEquation
>>> disEqn = DistanceEquation(var)
```

The `advectionEquation` is constructed.

```
>>> from fipy.models.levelSet.advection.advectionEquation import AdvectionEquation
>>> advEqn = AdvectionEquation(var, advectionCoeff = velocity)
```

An `Iterator` object is constructed.

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((advEqn,))
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var = var, palette = 'rainbow.gp', minVal = -10., maxVal = 10.)
...     viewer.plot()
...     disEqn.solve()
...     for step in range(steps):
...         it.timestep(dt = timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> disEqn.solve()
>>> for step in range(steps):
...     it.timestep(dt = timeStepDuration)
>>> import Numeric
>>> x = Numeric.array(mesh.getCellCenters()[:,0])
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
>>> answer = Numeric.where(x < distanceTravelled, x[0] - interfacePosition, answer)
>>> Numeric.allclose(answer, Numeric.array(var), atol = 1e-10)
1
```

8.4 Module examples.levelSet.advection.circle.input

This example first imposes a circular distance function:

$$\phi\left(x, y\right) = \left[\left(x - \frac{L}{2}\right)^2 + \left(y - \frac{L}{2}\right)^2\right]^{1/2} - \frac{L}{4}$$

The variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the `AdvectionTerm` preserves the `var` as a distance function. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> L = 1.
>>> nx = 25
>>> velocity = 1.
>>> cfl = 0.1
>>> velocity = 1.
>>> distanceToTravel = L / 10.
>>> radius = L / 4.
>>> dx = L / nx
>>> timeStepDuration = cfl * dx / velocity
>>> steps = int(distanceToTravel / dx / cfl)
```

Construct the mesh.

```
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dx, nx = nx, ny = nx)
```

Construct a `distanceVariable` object.

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> var = DistanceVariable(
...     name = 'level set variable',
...     mesh = mesh,
...     value = 1.)
```

Initialise the `distanceVariable` to be a circular distance function.

```
>>> import Numeric
>>> initialArray = Numeric.sqrt((mesh.getCellCenters()[:,0] - L / 2.)**2 +
...                             (mesh.getCellCenters()[:,1] - L / 2.)**2) - radius
>>> var.setValue(initialArray)
```

The `advectionEquation` is constructed.

```
>>> from fipy.models.levelSet.advection.advectionEquation import AdvectionEquation
>>> advEqn = AdvectionEquation(
...     var,
...     advectionCoeff = velocity)
```

An `Iterator` object is constructed.

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((advEqn,))
```

The problem can then be solved by executing a serious of time steps.

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var = var, palette = 'rainbow.gp', minVal = -radius,
...                               maxVal = radius)
...     viewer.plot()
...     for step in range(steps):
...         it.timestep(dt = timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following commands.

```
>>> for step in range(steps):
...     it.timestep(dt = timeStepDuration)
>>> x = Numeric.array(mesh.getCellCenters())
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = Numeric.where(answer < 0., -1001., answer)
>>> solution = Numeric.where(answer < 0., -1001., Numeric.array(var))
>>> Numeric.allclose(answer, solution, atol = 4.7e-3)
1
```

If the `AdvectionEquation` is built with the `HigherOrderAdvectionTerm` the result is more accurate,

```
>>> var.setValue(initialArray)
>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation import HigherOrderAdvectionEq
>>> advEqn = HigherOrderAdvectionEquation(
...     var,
...     advectionCoeff = velocity)
>>> it = Iterator((advEqn,))
>>> for step in range(steps):
...     it.timestep(dt = timeStepDuration)
>>> solution = Numeric.where(answer < 0., -1001., Numeric.array(var))
>>> Numeric.allclose(answer, solution, atol = 1.02e-3)
1
```

Superconformal Electrodeposition Example

Electroplating is a deposition method widely used to fill high-aspect ratio features without seams or voids through the process of superconformal deposition, also called “superfill.” This process has been demonstrated to depend critically on the inclusion of additives in the electrolyte. Recent publications propose “Curvature Enhanced Accelerator Coverage” (CEAC) as the mechanism behind the superfilling process [4]. In this mechanism, molecules that accelerate local metal deposition displace molecules that inhibit local metal deposition on the metal/electrolyte interface. For electrolytes that yield superconformal filling of fine features, this buildup happens relatively slowly because the concentration of accelerator species is much more dilute compared to the inhibitor species in the electrolyte. The mechanism that leads to the increased rate of metal deposition along the bottom of the filling trench is the concurrent local increase of the accelerator coverage due to decreasing local surface area, which scales with the local curvature (hence the name of the mechanism).

8.5 Module examples.levelSet.electroChem.input

This input file is a demonstration of the use of FiPy for modeling copper electroplating. The material properties and experimental parameters used are roughly those that have been previously published [17]. To run this example from the base fipy directory type:

```
$ examples/diffusion/steadyState/mesh1D/input.py
```

at the command line. The simulation took about 5 minutes on a computer with a 2GHz Athlon CPU. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. The Gist package is required to view the results as the simulation is being executed (see the installation guide in chapter 2). The following is an explicit explanation of the input commands required to set up and run the problem. At the top of the file all the parameter values are set. Their use will be explained during the instantiation of various objects.

The following parameters (all in S.I. units) represent,

physical constants,

```
>>> faradaysConstant = 9.6e4
>>> gasConstant = 8.314
>>> transferCoefficient = 0.5
```

properties associated with the accelerator species,

```
>>> rateConstant = 1.76
>>> overpotentialDependence = -245e-6
>>> acceleratorDiffusionCoefficient = 1e-9
>>> siteDensity = 9.8e-6
```

properties of the cupric ions,

```
>>> atomicVolume = 7.1e-6,
>>> charge = 2
>>> metalDiffusionCoefficient = 5.6e-10
```

parameters dependent on experimental constraints,

```
>>> temperature = 298.
>>> overpotential = -0.3
>>> bulkMetalConcentration = 250.
>>> bulkAcceleratorConcentration = 5e-3
>>> initialAcceleratorCoverage = 0.
```

parameters obtained from experiments on flat copper electrodes,

```
>>> constantCurrentDensity = 0.26
>>> acceleratorDependenceCurrentDensity = 45.
```

general simulation control parameters,

```
>>> numberOfSteps = 300
>>> cflNumber = 0.2
>>> numberOfCellsInNarrowBand = 10
>>> cellsBelowTrench = 10
>>> cellSize = 0.1e-7
```

parameters required for a trench geometry,

```
>>> trenchDepth = 0.5e-6
>>> aspectRatio = 2.
>>> trenchSpacing = 0.6e-6
>>> boundaryLayerDepth = 0.3e-6
```

The hydrodynamic boundary layer depth (`boundaryLayerDepth`) is intentionally small in this example to keep the mesh at a reasonable size.

Build the mesh:

```
>>> yCells = cellsBelowTrench + int((trenchDepth + boundaryLayerDepth) / cellSize)
>>> xCells = int(trenchSpacing / 2 / cellSize)
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx = cellSize,
...               dy = cellSize,
...               nx = xCells,
...               ny = yCells)
```

A `distanceVariable` object, ϕ, is required to store the position of the interface (at $\phi = 0$). A `distanceEquation` object is used to solve the `distanceVariable` so that it has the value of a distance function (i.e. holds the distance at any point in the mesh from the electrolyte/metal interface). The `distanceEquation` object solves the equation $|\nabla\phi| = 1$.

Firstly, create the ϕ variable:

```
>>> from fipy.models.levelSet.distanceFunction.distanceVariable import DistanceVariable
>>> distanceVar = DistanceVariable(
...     name = 'distance variable',
...     mesh = mesh,
...     value = -1.)
```

This is initially set to -1 everywhere. The electrolyte region will be the positive region of the domain while the metal region will be negative. Create a function for returning cells that lie in the electrolyte region (positive region).

```
>>> bottomHeight = cellsBelowTrench * cellSize
>>> trenchHeight = bottomHeight + trenchDepth
>>> trenchWidth = trenchDepth / aspectRatio
>>> sideWidth = (trenchSpacing - trenchWidth) / 2
>>> def electrolyteFunc(cell):
...     x,y = cell.getCenter()
...     if y > trenchHeight:
...         return 1
...     elif y < bottomHeight:
...         return 0
...     elif x < sideWidth:
...         return 0
...     else:
...         return 1
```

Get the positive cells by passing the function,

```
>>> electrolyteCells = mesh.getCells(electrolyteFunc)
```

and set those cells to have a positive value

```
>>> distanceVar.setValue(1., electrolyteCells)
```

Next set up the `distanceEquation` and solve to set ϕ to be a distance function.

```
>>> terminationValue = numberOfCellsInNarrowBand / 2 * cellSize
>>> from fipy.models.levelSet.distanceFunction.distanceEquation import DistanceEquation
>>> distanceEquation = DistanceEquation(distanceVar, terminationValue = terminationValue)
>>> distanceEquation.solve(terminationValue = 1e+10)
```

The `distanceVariable` has now been created to mark the interface. Some other variables need to be created that govern the concentrations of various species. Create the accelerator surfactant coverage, θ, variable. This variable influences the deposition rate.

```
>>> from fipy.models.levelSet.surfactant.surfactantVariable import SurfactantVariable
>>> acceleratorVar = SurfactantVariable(
...     name = "accelerator variable",
...     value = initialAcceleratorCoverage,
...     distanceVar = distanceVar)
```

Create the bulk accelerator concentration, c_θ, in the electrolyte,

```
>>> from fipy.variables.cellVariable import CellVariable
>>> bulkAcceleratorVar = CellVariable(
...     name = 'bulk accelerator variable',
...     mesh = mesh,
...     value = bulkAcceleratorConcentration)
```

Create the bulk metal ion concentration, c_m, in the electrolyte.

```
>>> from fipy.variables.cellVariable import CellVariable
>>> metalVar = CellVariable(
...     name = 'metal variable',
...     mesh = mesh,
...     value = bulkMetalConcentration)
```

The following commands build the `depositionRateVariable`, v. The `depositionRateVariable` is given by the following equation.

$$v = \frac{i\Omega}{nF}$$

where Ω is the metal atomic volume, n is the metal ion charge and F is Faraday's constant. The current density is given by

$$i = i_0 \frac{c_m^i}{c_m^\infty} \exp\left(\frac{-\alpha F}{RT}\eta\right)$$

where c_m^i is the metal ion concentration in the bulk at the interface, c_m^∞ is the far-field bulk concentration of metal ions, α is the transfer coefficient, R is the gas constant, T is the temperature and η is the overpotential. The exchange current density is an empirical function of accelerator coverage,

$$i_0(\theta) = b_0 + b_1\theta$$

The commands needed to build this equation are,

```
>>> expoConstant = -transferCoefficient * faradaysConstant / gasConstant / temperature
>>> tmp = acceleratorDependenceCurrentDensity * acceleratorVar.getInterfaceVar()
>>> exchangeCurrentDensity = constantCurrentDensity + tmp
>>> import Numeric
>>> expo = Numeric.exp(expoConstant * overpotential)
>>> currentDensity = exchangeCurrentDensity * metalVar / bulkMetalConcentration * expo
>>> depositionRateVariable = currentDensity * atomicVolume / charge / faradaysConstant
```

Build the extension velocity variable v_{ext}. The extension velocity uses the `extensionEquation` to spread the velocity at the interface to the rest of the domain.

```
>>> extensionVelocityVariable = CellVariable(
...     name = 'extension velocity',
...     mesh = mesh,
...     value = depositionRateVariable)
```

Using the variables created above the governing equations will be built. The governing equation for surfactant conservation is given by,

$$\dot{\theta} = Jv\theta + kc_\theta^i(1-\theta)$$

where θ is the coverage of accelerator at the interface, J is the curvature of the interface, v is the normal velocity of the interface, c_θ^i is the concentration of accelerator in the bulk at the interface. The value k is given by an empirical function of overpotential,

$$k = k_0 + k_3 \eta^3$$

The above equation is represented by the `AdsorbingSurfactantEquation` in FiPy:

```
>>> from fipy.models.levelSet.surfactant.adsorbingSurfactantEquation \
...             import AdsorbingSurfactantEquation
>>> surfactantEquation = AdsorbingSurfactantEquation(
...     acceleratorVar,
...     distanceVar,
...     bulkAcceleratorConcentration,
...     rateConstant + overpotentialDependence * overpotential**3)
```

The variable ϕ is advected by the `advectionEquation` given by,

$$\frac{\partial \phi}{\partial t} + v_{\text{ext}} |\nabla \phi| = 0$$

and is set up with the following commands:

```
>>> from fipy.models.levelSet.advection.higherOrderAdvectionEquation \
...                import HigherOrderAdvectionEquation
>>> advectionEquation = HigherOrderAdvectionEquation(
...     distanceVar,
...     advectionCoeff = extensionVelocityVariable)
```

The `extensionEquation` extends the interface velocity v to v_{ext} throughout the whole domain using $\nabla \phi \cdot \nabla v_{\text{ext}} = 0$. The `extensionEquation` is set up with the following commands.

```
>>> from fipy.models.levelSet.distanceFunction.extensionEquation import ExtensionEquation
>>> extensionEquation = ExtensionEquation(
...     distanceVar,
...     extensionVelocityVariable,
...     terminationValue = terminationValue)
```

The diffusion of metal ions from the far field to the interface is governed by,

$$\frac{\partial c_m}{\partial t} = \nabla \cdot D \nabla c_m$$

where,

$$D = \begin{cases} D_m & \text{when } \phi > 0, \\ 0 & \text{when } \phi \le 0 \end{cases}$$

The following boundary condition applies at $\phi = 0$,

$$D\hat{n} \cdot \nabla c = \frac{v}{\Omega}.$$

The `MetalIonDiffusionEquation` is set up with the following commands.

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.models.levelSet.electroChem.metalIonDiffusionEquation \
...                      import MetalIonDiffusionEquation
>>> metalEquation = MetalIonDiffusionEquation(
...     metalVar,
...     distanceVar = distanceVar,
...     depositionRate = depositionRateVariable,
...     diffusionCoeff = metalDiffusionCoefficient,
...     metalIonAtomicVolume = atomicVolume,
...     boundaryConditions = (
...         FixedValue(
...             mesh.getFacesTop(),
...             bulkMetalConcentration
...         ),
...     )
... )
```

The `SurfactantBulkDiffusionEquation` solves the bulk diffusion of a species with a source term for the jump from the bulk to an interface. The governing equation is given by,

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

where,

$$D = \begin{cases} D_\theta & \text{when } \phi > 0 \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The jump condition at the interface is defined by Langmuir adsorption. Langmuir adsorption essentially states that the ability for a species to jump from an electrolyte to an interface is proportional to the concentration in the electrolyte, available site density and a jump coefficient. The boundary condition at $\phi = 0$ is given by,

$$D\hat{n} \cdot \nabla c = -kc(1 - \theta).$$

The `SurfactantBulkDiffusionEquation` is set up with the following commands.

```
>>> from fipy.models.levelSet.surfactant.surfactantBulkDiffusionEquation \
...                 import SurfactantBulkDiffusionEquation
>>> bulkAcceleratorEquation = SurfactantBulkDiffusionEquation(
...     bulkAcceleratorVar,
...     distanceVar = distanceVar,
...     surfactantVar = acceleratorVar,
...     diffusionCoeff = acceleratorDiffusionCoefficient,
...     rateConstant = rateConstant * siteDensity,
...         boundaryConditions = (
...         FixedValue(
...             mesh.getFacesTop(),
...             bulkAcceleratorConcentration
...         ),
...     )
... )
```

The equations are now given to an `Iterator` object in the order that they will be solved.

```
>>> from fipy.iterators.iterator import Iterator
>>> iterator = Iterator((extensionEquation,
...                      advectionEquation,
...                      surfactantEquation,
...                      metalEquation,
...                      bulkAcceleratorEquation))
```

The function below is constructed to encapsulate the creation of the viewers.

```
>>> def buildViewers():
...    from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...    resolution = 3
...    cells = yCells * 2**(resolution-1)
...    return (
...        Grid2DGistViewer(
...            var = distanceVar,
...            minVal = -1e-8,
...            maxVal = 1e-8,
...            grid = 0,
...            limits = (0, cells, 0, cells),
...            dpi = 100,
...            resolution = resolution),
...        Grid2DGistViewer(
...            var = acceleratorVar.getInterfaceVar(),
...            grid = 0,
...            limits = (0, cells, 0, cells),
...            dpi = 100,
...            resolution = resolution))
```

The `levelSetUpdateFrequency` defines how often to call the `distanceEquation` to reinitialize the `distanceVariable` to a distance function.

```
>>> levelSetUpdateFrequency = int(0.8 * terminationValue / cellSize / cflNumber)
```

The following loop runs for `numberOfSteps` time steps. The time step is calculated with the CFL number and the maximum extension velocity.

```
>>> if __name__ == '__main__':
...     viewers = buildViewers()
...     for step in range(numberOfSteps):
```

```
...         if step % levelSetUpdateFrequency == 0:
...             distanceEquation.solve()
...         extensionVelocityVariable.setValue(Numeric.array(depositionRateVariable))
...         argmax = Numeric.argmax(extensionVelocityVariable)
...         iterator.timestep(dt = cflNumber * cellSize / extensionVelocityVariable[argmax])
...         for viewer in viewers:
...             viewer.plot()
...     raw_input('finished')
```

The following is a short test case. It uses saved data from a simulation with 5 time steps. It is not a test for accuracy but a way to tell if something has changed or been broken.

```
>>> for i in range(5):
...     iterator.timestep(dt = 0.1)

>>> import os
>>> testFile = 'test.gz'
>>> import examples.levelSet.electroChem
>>> gzfile = 'gunzip --fast -c < %s/%s'
>>> gzfile = gzfile%(examples.levelSet.electroChem.__path__[0], testFile)
>>> filestream=os.popen(gzfile,'r')
>>> import cPickle
>>> testData = cPickle.load(filestream)
>>> filestream.close()

>>> Numeric.allclose(Numeric.array(acceleratorVar), testData)
1
```

Chapter 9

Cahn-Hilliard Examples

9.1 Module examples.cahnHilliard.inputTanh1D

This example solves the Cahn-Hilliard equation given by:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

where the free energy functional is given by,

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2.$$

We solve the problem on a 1D mesh

```
>>> L = 40.
>>> nx = 10000
>>> ny = 1
>>> dx = L / nx
>>> dy = 1.
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(dx, dy, nx, ny)
```

and create the solution variable

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "phase field",
...     mesh = mesh,
...     value = 1)
>>> var.setValue(1, cells = mesh.getCells(lambda cell: cell.getCenter()[0] > L / 2))
```

The boundary conditions for this problem are

$$\left.\begin{aligned} \phi &= \frac{1}{2} \\ \frac{\partial^3 \phi}{\partial x^3} &= 0 \end{aligned}\right\} \qquad \text{on } x = 0$$

and

$$\left.\begin{aligned} \phi &= 1 \\ \frac{\partial^2 \phi}{\partial x^2} &= 0 \end{aligned}\right\} \qquad \text{on } x = L$$

or

```
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.nthOrderBoundaryCondition \
...     import NthOrderBoundaryCondition
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesRight(), 1),
...     FixedValue(mesh.getFacesLeft(), .5),
...     NthOrderBoundaryCondition(mesh.getFacesLeft(), 0, 2),
...     NthOrderBoundaryCondition(mesh.getFacesRight(), 0, 3))
```

Using

```
>>> parameters={
...     'asq' : 1.0,
...     'epsilon' : 1,
...     'diffusionCoeff' : 1
... }
```

we create the Cahn-Hilliard equation object

```
>>> from fipy.solvers.linearLUSolver import LinearLUSolver
>>> from fipy.models.cahnHilliard.cahnHilliardEquation import CahnHilliardEquation
>>> eqch= CahnHilliardEquation(
...     var,
...     parameters = parameters,
...     solver = LinearLUSolver(
...         tolerance = 1e-15,
...         steps = 100),
...     boundaryConditions = boundaryConditions
... )
```

The solution to this 1D problem over an infinite domain is given by,

$$\phi(x) = \frac{1}{1 + \exp\left(-\frac{a}{\epsilon} x\right)}$$

or

```
>>> import Numeric
>>> a = Numeric.sqrt(parameters['asq'])
>>> answer = 1 / (1 +
...     Numeric.exp(-a * (mesh.getCellCenters()[:,0] - L / 2) / parameters['epsilon']))
```

If we are running interactively, we create a viewer to see the results

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
...     viewer = Grid2DGistViewer(var, minVal=0., maxVal=1.0, palette = 'rainbow.gp')
...     viewer.plot()
```

We iterate the solution to equilibrium and, if we are running interactively, we update the display and output data about the progression of the solution

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eqch,))
>>> dexp=-5
>>> for step in range(100):
...     dt = Numeric.exp(dexp)
...     dt = min(10,dt)
...     dexp += 0.5
...     it.timestep(dt = dt)
...     if __name__ == '__main__':
...         diff = abs(answer - Numeric.array(var))
...         maxarg = Numeric.argmax(diff)
...         print 'maximum error:',diff[maxarg]
...         print 'element id:',maxarg
...         print 'value at element ',maxarg,' is ',var[maxarg]
...
...         viewer.plot()
```

We compare the analytical solution with the numerical result,

```
>>> Numeric.allclose(var, answer, atol = 1e-2)
1
>>> Numeric.allclose(var, answer, atol = 1e-3)
1
>>> Numeric.allclose(var, answer, atol = 1e-4)
0
>>> Numeric.allclose(var, answer, atol = 1e-5)
0
```

Bibliography

[1] The Python Programming Language, URL http://www.python.org/. 5

[2] W. J. Boettinger, J. A. Warren, C. Beckermann, and A. Karma, "Phase-field simulation of solidification". *Annual Review of Materials Research*, **32**, (2002) 163–194, URL http://arjournals.annualreviews.org/doi/abs/10.1146/annurev.matsci.32.101901.155803. 5

[3] G. B. McFadden, "Phase-field models of solidification". *Contemporary Mathematics*, **306**, (2002) 107–145. 5

[4] D. Josell, D. Wheeler, W. H. Huber, and T. P. Moffat, "Superconformal Electrodeposition in Submicron Features". *Physical Review Letters*, **87**(1), (2001) 016102, URL http://link.aps.org/abstract/PRL/v87/e016102. 5, 92

[5] Greg Ward, *Installing Python Modules*. URL http://docs.python.org/inst/. 9

[6] Guido van Rossum, *Python Tutorial*. URL http://docs.python.org/tut/. 14

[7] T. N. Croft, *Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent Reacting Flows*. Ph.D. thesis, University of Greenwich, 1998, URL http://www.gre.ac.uk/~ct02/research/thesis/main.html. 17, 20

[8] S. V. Patanker, *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980. 17

[9] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995. 17

[10] John W. Cahn and John E. Hilliard, "Free energy of a nonuniform system. I. Interfacial free energy". *Journal of Computational Physics*, **28**(2), (1958) 258–267. 18

[11] John W. Cahn, "Free energy of a nonuniform system. II. Thermodynamic basis". *Journal of Computational Physics*, **30**(5), (1959) 1121–1124. 18

[12] John W. Cahn and John E. Hilliard, "Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid". *Journal of Computational Physics*, **31**(3), (1959) 688–699. 18

[13] James A. Warren, Ryo Kobayashi, Alexander E. Lobkovsky, and W. Craig Carter, "Extending Phase Field Models of Solidification to Polycrystalline Materials". *Acta Materialia*, **51**(20), (2003) 6035–6058, URL http://dx.doi.org/10.1016/S1359-6454(03)00388-4. 57

[14] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden, "Phase field modeling of electrochemistry I: Equilibrium". *Physical Review E*, **69**, (2004) 021603, cond-mat/0308173, URL http://link.aps.org/abstract/PRE/v69/e021603. 72

[15] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden, "Phase field modeling of electrochemistry II: Kinetics". *Physical Review E*, **69**, (2004) 021604, cond-mat/0308179, URL http://link.aps.org/abstract/PRE/v69/e021604. 72

[16] J. A. Sethian, *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996. 85

[17] D. Wheeler, D. Josell, and T. P. Moffat, “Modeling Superconformal Electrodeposition Using The Level Set Method”. *Journal of The Electrochemical Society*, **150**(5), (2003) C302–C310. 92

Index

Contributors

Jon Guyer is a member of the research staff of the Metallurgy Division in the Materials Science and Engineering Laboratory at the National Institute of Standards and Technology. Jon's computational interests are in object-oriented design and in phase field modeling of electrochemistry.

Daniel Wheeler is a caveman. Daniel's interests are in numerical modeling, finite volume techniques, and level set treatments.

Jim Warren is a member of the research staff of the Metallurgy Division and the Director of the Center for Theoretical and Computational Materials Science of the Materials Science and Engineering Laboratory at the National Institute of Standards and Technology. Jim is interested in a variety of problems, including the phase field modeling of solidification, polycrystalline solids, and the electrochemical interface.

Alex Mont is a student at Montgomery Blair High School. Alex developed the *PyxViewer* and the *Gmsh* import and export modules.